Machine Learning: Algorithms and Applications

CSCI 370

Spring 2022
Professor: Marina Barsky

Basic Machine Learning Algorithms

Introduction to Data Mining (Second Edition)

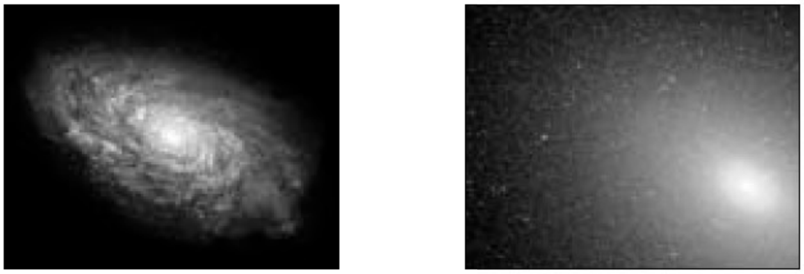
Pang-Ning Tan,
Michael Steinbach,
Anuj Karpatne,
Vipin Kumar

Selected Book Chapters

4

Classification: Basic Concepts, Decision Trees, and Model Evaluation

Classification, which is the task of assigning objects to one of several predefined categories, is a pervasive problem that encompasses many diverse applications. Examples include detecting spam email messages based upon the message header and content, categorizing cells as malignant or benign based upon the results of MRI scans, and classifying galaxies based upon their shapes (see Figure 4.1).



(a) A spiral galaxy.

(b) An elliptical galaxy.

Figure 4.1. Classification of galaxies. The images are from the NASA website.

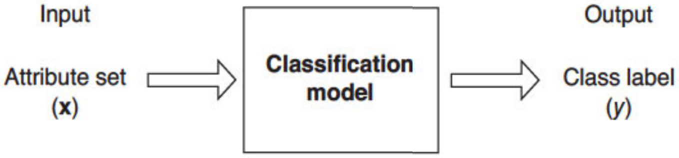


Figure 4.2. Classification as the task of mapping an input attribute set $\mathbf{x}$ into its class label y.

This chapter introduces the basic concepts of classification, describes some of the key issues such as model overfitting, and presents methods for evaluating and comparing the performance of a classification technique. While it focuses mainly on a technique known as decision tree induction, most of the discussion in this chapter is also applicable to other classification techniques, many of which are covered in Chapter 5.

4.1 Preliminaries

The input data for a classification task is a collection of records. Each record, also known as an instance or example, is characterized by a tuple $(\mathbf{x}, y)$, where $\mathbf{x}$ is the attribute set and y is a special attribute, designated as the class label (also known as category or target attribute). Table 4.1 shows a sample data set used for classifying vertebrates into one of the following categories: mammal, bird, fish, reptile, or amphibian. The attribute set includes properties of a vertebrate such as its body temperature, skin cover, method of reproduction, ability to fly, and ability to live in water. Although the attributes presented in Table 4.1 are mostly discrete, the attribute set can also contain continuous features. The class label, on the other hand, must be a discrete attribute. This is a key characteristic that distinguishes classification from **regression**, a predictive modeling task in which y is a continuous attribute. Regression techniques are covered in Appendix D.

Definition 4.1 (Classification). Classification is the task of learning a **target function** f that maps each attribute set $\mathbf{x}$ to one of the predefined class labels y.

The target function is also known informally as a **classification model**. A classification model is useful for the following purposes.

Descriptive Modeling A classification model can serve as an explanatory tool to distinguish between objects of different classes. For example, it would be useful—for both biologists and others—to have a descriptive model that

Table 4.1. The vertebrate data set.

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hiber-nates	Class Label
human	warm-blooded	hair	yes	no	no	yes	no	mammal
python	cold-blooded	scales	no	no	no	no	yes	reptile
salmon	cold-blooded	scales	no	yes	no	no	no	fish
whale	warm-blooded	hair	yes	yes	no	no	no	mammal
frog	cold-blooded	none	no	semi	no	yes	yes	amphibian
komodo dragon	cold-blooded	scales	no	no	no	yes	no	reptile
bat	warm-blooded	hair	yes	no	yes	yes	yes	mammal
pigeon	warm-blooded	feathers	no	no	yes	yes	no	bird
cat	warm-blooded	fur	yes	no	no	yes	no	mammal
leopard shark	cold-blooded	scales	yes	yes	no	no	no	fish
turtle	cold-blooded	scales	no	semi	no	yes	no	reptile
penguin	warm-blooded	feathers	no	semi	no	yes	no	bird
porcupine	warm-blooded	quills	yes	no	no	yes	yes	mammal
eel	cold-blooded	scales	no	yes	no	no	no	fish
salamander	cold-blooded	none	no	semi	no	yes	yes	amphibian

summarizes the data shown in Table 4.1 and explains what features define a vertebrate as a mammal, reptile, bird, fish, or amphibian.

Predictive Modeling A classification model can also be used to predict the class label of unknown records. As shown in Figure 4.2, a classification model can be treated as a black box that automatically assigns a class label when presented with the attribute set of an unknown record. Suppose we are given the following characteristics of a creature known as a gila monster:

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hiber-nates	Class Label
gila monster	cold-blooded	scales	no	no	no	yes	yes	?

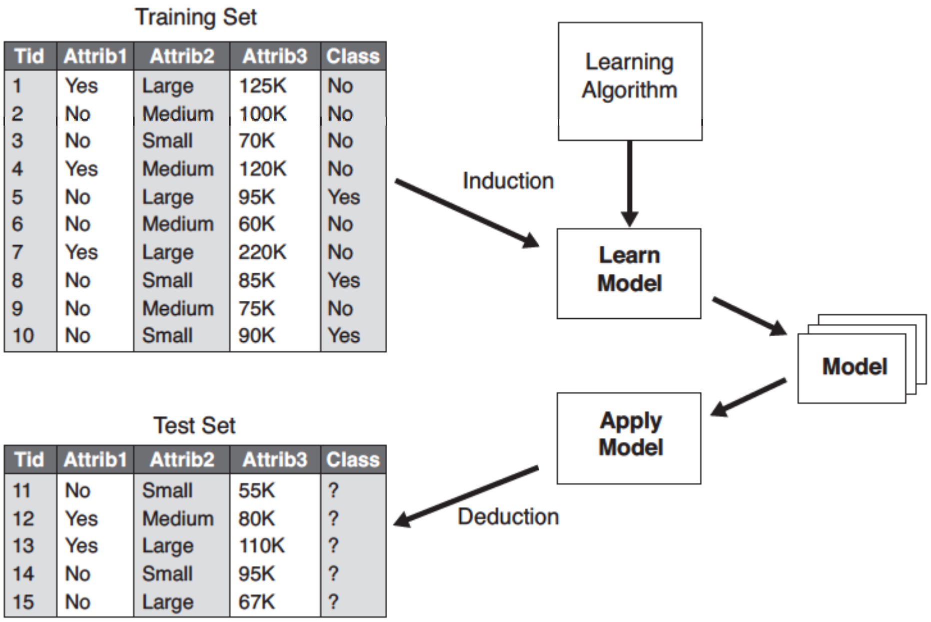
We can use a classification model built from the data set shown in Table 4.1 to determine the class to which the creature belongs.

Classification techniques are most suited for predicting or describing data sets with binary or nominal categories. They are less effective for ordinal categories (e.g., to classify a person as a member of high-, medium-, or low-income group) because they do not consider the implicit order among the categories. Other forms of relationships, such as the subclass–superclass relationships among categories (e.g., humans and apes are primates, which in turn, is a subclass of mammals) are also ignored. The remainder of this chapter focuses only on binary or nominal class labels.

4.2 General Approach to Solving a Classification Problem

A classification technique (or classifier) is a systematic approach to building classification models from an input data set. Examples include decision tree classifiers, rule-based classifiers, neural networks, support vector machines, and naïve Bayes classifiers. Each technique employs a **learning algorithm** to identify a model that best fits the relationship between the attribute set and class label of the input data. The model generated by a learning algorithm should both fit the input data well and correctly predict the class labels of records it has never seen before. Therefore, a key objective of the learning algorithm is to build models with good generalization capability; i.e., models that accurately predict the class labels of previously unknown records.

Figure 4.3 shows a general approach for solving classification problems. First, a **training set** consisting of records whose class labels are known must



Training Set

Tid	Attrib1	Attrib2	Attrib3	Class
1	Yes	Large	125K	No
2	No	Medium	100K	No
3	No	Small	70K	No
4	Yes	Medium	120K	No
5	No	Large	95K	Yes
6	No	Medium	60K	No
7	Yes	Large	220K	No
8	No	Small	85K	Yes
9	No	Medium	75K	No
10	No	Small	90K	Yes

Test Set

Tid	Attrib1	Attrib2	Attrib3	Class
11	No	Small	55K	?
12	Yes	Medium	80K	?
13	Yes	Large	110K	?
14	No	Small	95K	?
15	No	Large	67K	?

Figure 4.3. General approach for building a classification model.

Table 4.2. Confusion matrix for a 2-class problem.

		Predicted Class	
		$Class = 1$	$Class = 0$
Actual Class	$Class = 1$	f_{11}	f_{10}
	$Class = 0$	f_{01}	f_{00}

be provided. The training set is used to build a classification model, which is subsequently applied to the **test set**, which consists of records with unknown class labels.

Evaluation of the performance of a classification model is based on the counts of test records correctly and incorrectly predicted by the model. These counts are tabulated in a table known as a **confusion matrix**. Table 4.2 depicts the confusion matrix for a binary classification problem. Each entry f_{ij} in this table denotes the number of records from class i predicted to be of class j. For instance, f_{01} is the number of records from class 0 incorrectly predicted as class 1. Based on the entries in the confusion matrix, the total number of correct predictions made by the model is $(f_{11} + f_{00})$ and the total number of incorrect predictions is $(f_{10} + f_{01})$.

Although a confusion matrix provides the information needed to determine how well a classification model performs, summarizing this information with a single number would make it more convenient to compare the performance of different models. This can be done using a **performance metric** such as **accuracy**, which is defined as follows:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{f_{11} + f_{00}}{f_{11} + f_{10} + f_{01} + f_{00}}. \qquad (4.1)$$

Equivalently, the performance of a model can be expressed in terms of its **error rate**, which is given by the following equation:

$$\text{Error rate} = \frac{\text{Number of wrong predictions}}{\text{Total number of predictions}} = \frac{f_{10} + f_{01}}{f_{11} + f_{10} + f_{01} + f_{00}}. \qquad (4.2)$$

Most classification algorithms seek models that attain the highest accuracy, or equivalently, the lowest error rate when applied to the test set. We will revisit the topic of model evaluation in Section 4.5.

4.3 Decision Tree Induction

This section introduces a **decision tree** classifier, which is a simple yet widely used classification technique.

4.3.1 How a Decision Tree Works

To illustrate how classification with a decision tree works, consider a simpler version of the vertebrate classification problem described in the previous section. Instead of classifying the vertebrates into five distinct groups of species, we assign them to two categories: mammals and non-mammals.

Suppose a new species is discovered by scientists. How can we tell whether it is a mammal or a non-mammal? One approach is to pose a series of questions about the characteristics of the species. The first question we may ask is whether the species is cold- or warm-blooded. If it is cold-blooded, then it is definitely not a mammal. Otherwise, it is either a bird or a mammal. In the latter case, we need to ask a follow-up question: Do the females of the species give birth to their young? Those that do give birth are definitely mammals, while those that do not are likely to be non-mammals (with the exception of egg-laying mammals such as the platypus and spiny anteater).

The previous example illustrates how we can solve a classification problem by asking a series of carefully crafted questions about the attributes of the test record. Each time we receive an answer, a follow-up question is asked until we reach a conclusion about the class label of the record. The series of questions and their possible answers can be organized in the form of a decision tree, which is a hierarchical structure consisting of nodes and directed edges. Figure 4.4 shows the decision tree for the mammal classification problem. The tree has three types of nodes:

- A **root node** that has no incoming edges and zero or more outgoing edges.
- **Internal nodes**, each of which has exactly one incoming edge and two or more outgoing edges.
- **Leaf** or **terminal** nodes, each of which has exactly one incoming edge and no outgoing edges.

In a decision tree, each leaf node is assigned a class label. The **non-terminal** nodes, which include the root and other internal nodes, contain attribute test conditions to separate records that have different characteristics. For example, the root node shown in Figure 4.4 uses the attribute `Body`

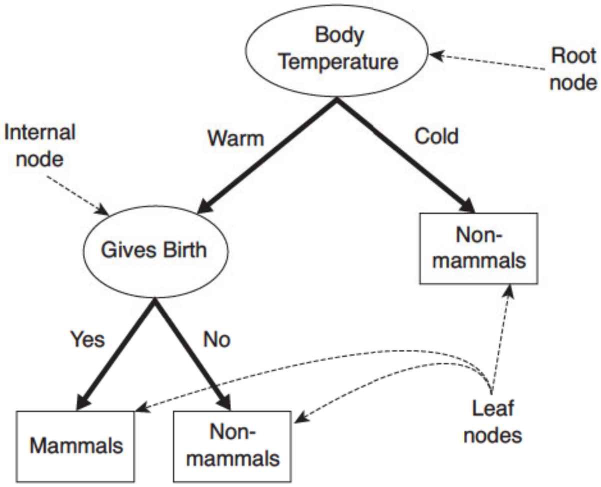


Figure 4.4. A decision tree for the mammal classification problem.

Temperature to separate warm-blooded from cold-blooded vertebrates. Since all cold-blooded vertebrates are non-mammals, a leaf node labeled **Non-mammals** is created as the right child of the root node. If the vertebrate is warm-blooded, a subsequent attribute, **Gives Birth**, is used to distinguish mammals from other warm-blooded creatures, which are mostly birds.

Classifying a test record is straightforward once a decision tree has been constructed. Starting from the root node, we apply the test condition to the record and follow the appropriate branch based on the outcome of the test. This will lead us either to another internal node, for which a new test condition is applied, or to a leaf node. The class label associated with the leaf node is then assigned to the record. As an illustration, Figure 4.5 traces the path in the decision tree that is used to predict the class label of a flamingo. The path terminates at a leaf node labeled **Non-mammals**.

4.3.2 How to Build a Decision Tree

In principle, there are exponentially many decision trees that can be constructed from a given set of attributes. While some of the trees are more accurate than others, finding the optimal tree is computationally infeasible because of the exponential size of the search space. Nevertheless, efficient algorithms have been developed to induce a reasonably accurate, albeit suboptimal, decision tree in a reasonable amount of time. These algorithms usually employ a greedy strategy that grows a decision tree by making a series of locally op-

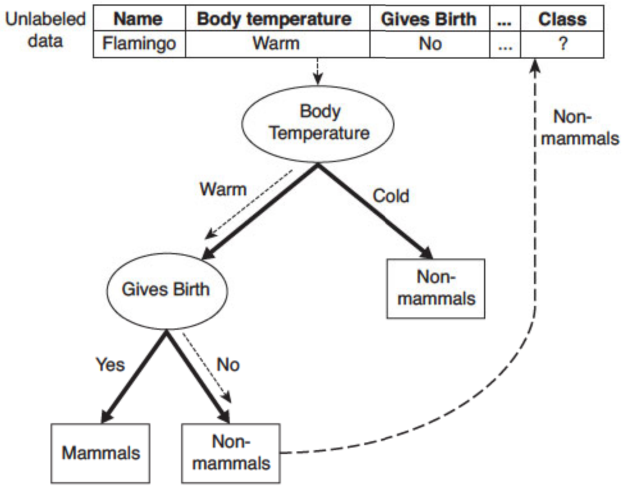


Figure 4.5. Classifying an unlabeled vertebrate. The dashed lines represent the outcomes of applying various attribute test conditions on the unlabeled vertebrate. The vertebrate is eventually assigned to the `Non-mammal` class.

timum decisions about which attribute to use for partitioning the data. One such algorithm is **Hunt's algorithm**, which is the basis of many existing decision tree induction algorithms, including ID3, C4.5, and CART. This section presents a high-level discussion of Hunt's algorithm and illustrates some of its design issues.

Hunt's Algorithm

In Hunt's algorithm, a decision tree is grown in a recursive fashion by partitioning the training records into successively purer subsets. Let D_t be the set of training records that are associated with node t and $y = \{y_1, y_2, \ldots, y_c\}$ be the class labels. The following is a recursive definition of Hunt's algorithm.

Step 1: If all the records in D_t belong to the same class y_t, then t is a leaf node labeled as y_t.

Step 2: If D_t contains records that belong to more than one class, an **attribute test condition** is selected to partition the records into smaller subsets. A child node is created for each outcome of the test condition and the records in D_t are distributed to the children based on the outcomes. The algorithm is then recursively applied to each child node.

	binary	categorical	continuous	class
Tid	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Figure 4.6. Training set for predicting borrowers who will default on loan payments.

To illustrate how the algorithm works, consider the problem of predicting whether a loan applicant will repay her loan obligations or become delinquent, subsequently defaulting on her loan. A training set for this problem can be constructed by examining the records of previous borrowers. In the example shown in Figure 4.6, each record contains the personal information of a borrower along with a class label indicating whether the borrower has defaulted on loan payments.

The initial tree for the classification problem contains a single node with class label `Defaulted = No` (see Figure 4.7(a)), which means that most of the borrowers successfully repaid their loans. The tree, however, needs to be refined since the root node contains records from both classes. The records are subsequently divided into smaller subsets based on the outcomes of the `Home Owner` test condition, as shown in Figure 4.7(b). The justification for choosing this attribute test condition will be discussed later. For now, we will assume that this is the best criterion for splitting the data at this point. Hunt's algorithm is then applied recursively to each child of the root node. From the training set given in Figure 4.6, notice that all borrowers who are home owners successfully repaid their loans. The left child of the root is therefore a leaf node labeled `Defaulted = No` (see Figure 4.7(b)). For the right child, we need to continue applying the recursive step of Hunt's algorithm until all the records belong to the same class. The trees resulting from each recursive step are shown in Figures 4.7(c) and (d).

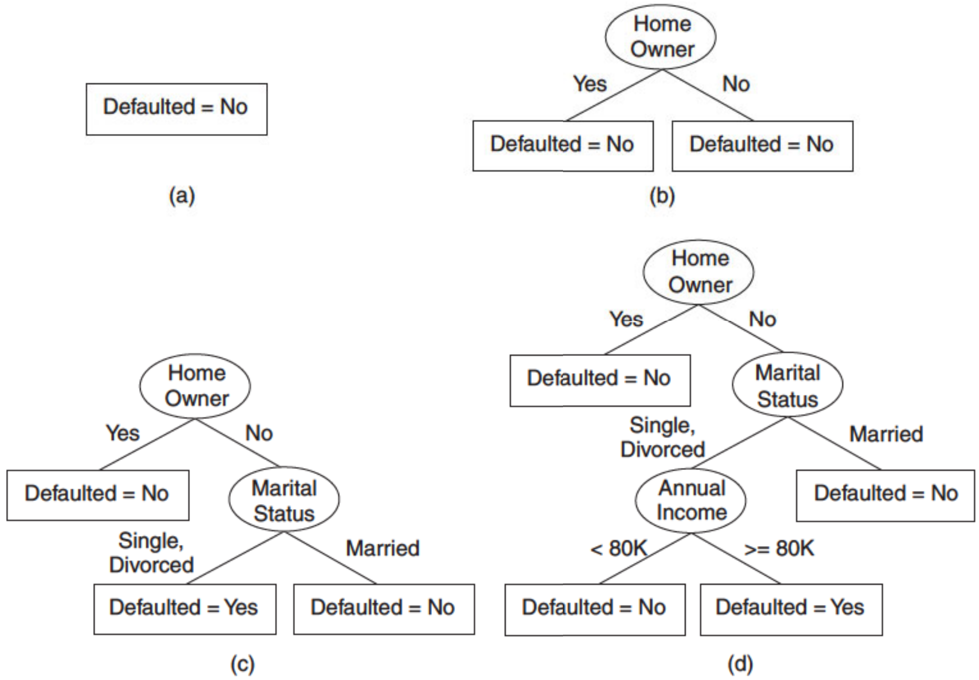


Figure 4.7. Hunt's algorithm for inducing decision trees.

Hunt's algorithm will work if every combination of attribute values is present in the training data and each combination has a unique class label. These assumptions are too stringent for use in most practical situations. Additional conditions are needed to handle the following cases:

1. It is possible for some of the child nodes created in Step 2 to be empty; i.e., there are no records associated with these nodes. This can happen if none of the training records have the combination of attribute values associated with such nodes. In this case the node is declared a leaf node with the same class label as the majority class of training records associated with its parent node.

2. In Step 2, if all the records associated with D_t have identical attribute values (except for the class label), then it is not possible to split these records any further. In this case, the node is declared a leaf node with the same class label as the majority class of training records associated with this node.

Design Issues of Decision Tree Induction

A learning algorithm for inducing decision trees must address the following two issues.

1. **How should the training records be split?** Each recursive step of the tree-growing process must select an attribute test condition to divide the records into smaller subsets. To implement this step, the algorithm must provide a method for specifying the test condition for different attribute types as well as an objective measure for evaluating the goodness of each test condition.

2. **How should the splitting procedure stop?** A stopping condition is needed to terminate the tree-growing process. A possible strategy is to continue expanding a node until either all the records belong to the same class or all the records have identical attribute values. Although both conditions are sufficient to stop any decision tree induction algorithm, other criteria can be imposed to allow the tree-growing procedure to terminate earlier. The advantages of early termination will be discussed later in Section 4.4.5.

4.3.3 Methods for Expressing Attribute Test Conditions

Decision tree induction algorithms must provide a method for expressing an attribute test condition and its corresponding outcomes for different attribute types.

Binary Attributes The test condition for a binary attribute generates two potential outcomes, as shown in Figure 4.8.

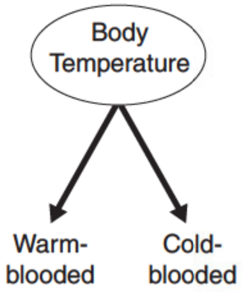


Figure 4.8. Test condition for binary attributes.

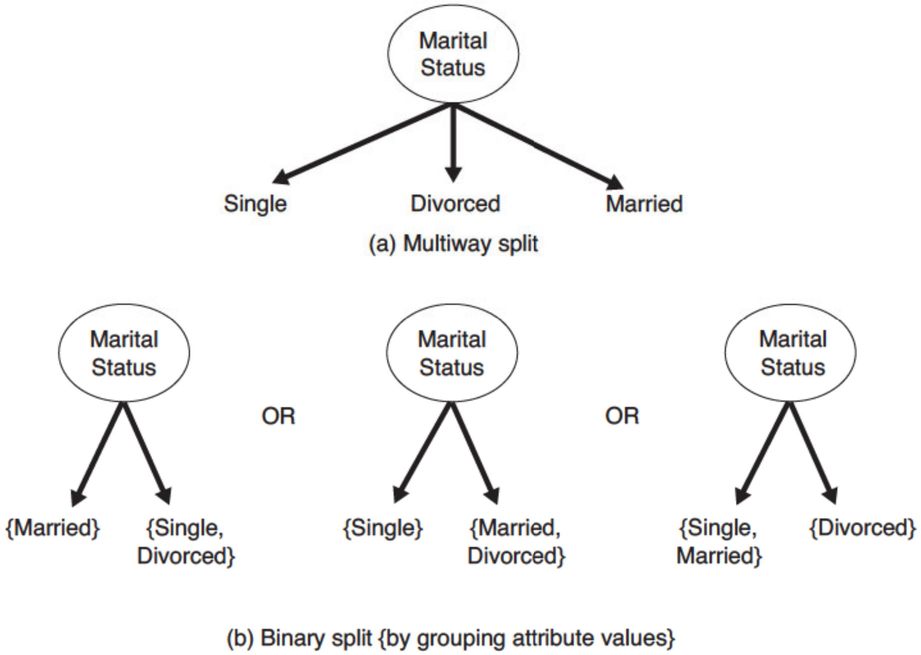


Figure 4.9. Test conditions for nominal attributes.

Nominal Attributes Since a nominal attribute can have many values, its test condition can be expressed in two ways, as shown in Figure 4.9. For a multiway split (Figure 4.9(a)), the number of outcomes depends on the number of distinct values for the corresponding attribute. For example, if an attribute such as marital status has three distinct values—single, married, or divorced—its test condition will produce a three-way split. On the other hand, some decision tree algorithms, such as CART, produce only binary splits by considering all $2^{k-1} - 1$ ways of creating a binary partition of k attribute values. Figure 4.9(b) illustrates three different ways of grouping the attribute values for marital status into two subsets.

Ordinal Attributes Ordinal attributes can also produce binary or multiway splits. Ordinal attribute values can be grouped as long as the grouping does not violate the order property of the attribute values. Figure 4.10 illustrates various ways of splitting training records based on the `Shirt Size` attribute. The groupings shown in Figures 4.10(a) and (b) preserve the order among the attribute values, whereas the grouping shown in Figure 4.10(c) violates this property because it combines the attribute values `Small` and `Large` into

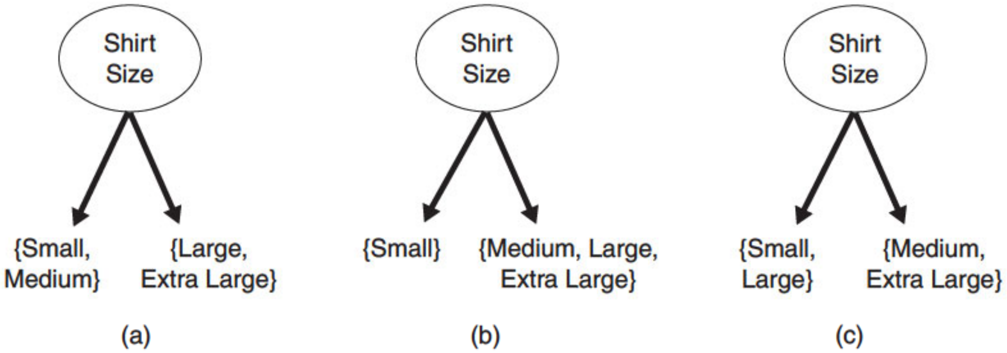


Figure 4.10. Different ways of grouping ordinal attribute values.

the same partition while `Medium` and `Extra Large` are combined into another partition.

Continuous Attributes For continuous attributes, the test condition can be expressed as a comparison test $(A < v)$ or $(A \geq v)$ with binary outcomes, or a range query with outcomes of the form $v_i \leq A < v_{i+1}$, for $i = 1, \ldots, k$. The difference between these approaches is shown in Figure 4.11. For the binary case, the decision tree algorithm must consider all possible split positions v, and it selects the one that produces the best partition. For the multiway split, the algorithm must consider all possible ranges of continuous values. One approach is to apply the discretization strategies described in Section 2.3.6 on page 57. After discretization, a new ordinal value will be assigned to each discretized interval. Adjacent intervals can also be aggregated into wider ranges as long as the order property is preserved.

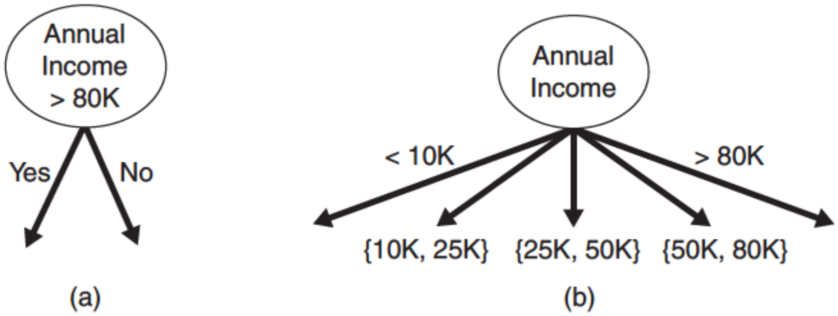


Figure 4.11. Test condition for continuous attributes.

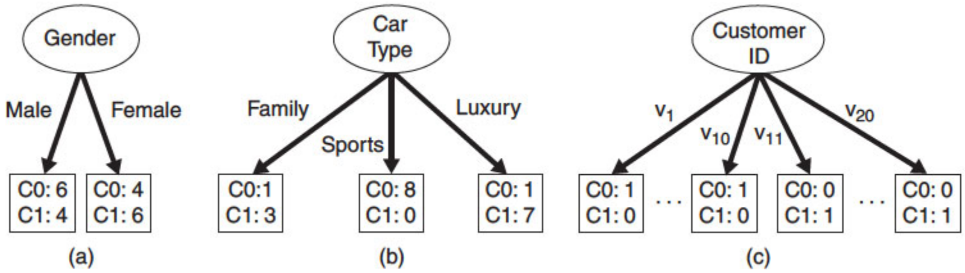


Figure 4.12. Multiway versus binary splits.

4.3.4 Measures for Selecting the Best Split

There are many measures that can be used to determine the best way to split the records. These measures are defined in terms of the class distribution of the records before and after splitting.

Let $p(i|t)$ denote the fraction of records belonging to class i at a given node t. We sometimes omit the reference to node t and express the fraction as p_i. In a two-class problem, the class distribution at any node can be written as (p_0, p_1), where $p_1 = 1 - p_0$. To illustrate, consider the test conditions shown in Figure 4.12. The class distribution before splitting is $(0.5, 0.5)$ because there are an equal number of records from each class. If we split the data using the `Gender` attribute, then the class distributions of the child nodes are $(0.6, 0.4)$ and $(0.4, 0.6)$, respectively. Although the classes are no longer evenly distributed, the child nodes still contain records from both classes. Splitting on the second attribute, `Car Type`, will result in purer partitions.

The measures developed for selecting the best split are often based on the degree of impurity of the child nodes. The smaller the degree of impurity, the more skewed the class distribution. For example, a node with class distribution $(0, 1)$ has zero impurity, whereas a node with uniform class distribution $(0.5, 0.5)$ has the highest impurity. Examples of impurity measures include

$$\text{Entropy}(t) = -\sum_{i=0}^{c-1} p(i|t) \log_2 p(i|t), \tag{4.3}$$

$$\text{Gini}(t) = 1 - \sum_{i=0}^{c-1} [p(i|t)]^2, \tag{4.4}$$

$$\text{Classification error}(t) = 1 - \max_i [p(i|t)], \tag{4.5}$$

where c is the number of classes and $0 \log_2 0 = 0$ in entropy calculations.

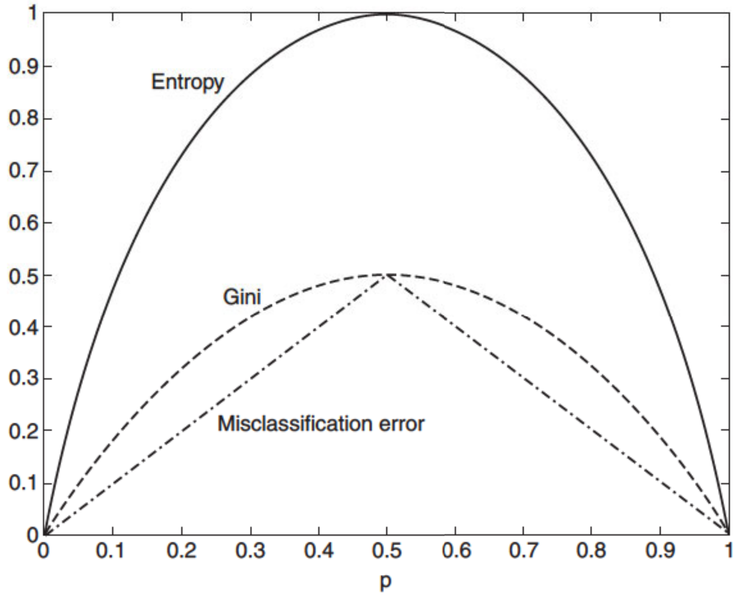


Figure 4.13. Comparison among the impurity measures for binary classification problems.

Figure 4.13 compares the values of the impurity measures for binary classification problems. p refers to the fraction of records that belong to one of the two classes. Observe that all three measures attain their maximum value when the class distribution is uniform (i.e., when $p = 0.5$). The minimum values for the measures are attained when all the records belong to the same class (i.e., when p equals 0 or 1). We next provide several examples of computing the different impurity measures.

Node N_1	Count
Class=0	0
Class=1	6

$$\begin{aligned}\text{Gini} &= 1 - (0/6)^2 - (6/6)^2 = 0\\ \text{Entropy} &= -(0/6)\log_2(0/6) - (6/6)\log_2(6/6) = 0\\ \text{Error} &= 1 - \max[0/6, 6/6] = 0\end{aligned}$$

Node N_2	Count
Class=0	1
Class=1	5

$$\begin{aligned}\text{Gini} &= 1 - (1/6)^2 - (5/6)^2 = 0.278\\ \text{Entropy} &= -(1/6)\log_2(1/6) - (5/6)\log_2(5/6) = 0.650\\ \text{Error} &= 1 - \max[1/6, 5/6] = 0.167\end{aligned}$$

Node N_3	Count
Class=0	3
Class=1	3

$$\begin{aligned}\text{Gini} &= 1 - (3/6)^2 - (3/6)^2 = 0.5\\ \text{Entropy} &= -(3/6)\log_2(3/6) - (3/6)\log_2(3/6) = 1\\ \text{Error} &= 1 - \max[3/6, 3/6] = 0.5\end{aligned}$$

The preceding examples, along with Figure 4.13, illustrate the consistency among different impurity measures. Based on these calculations, node N_1 has the lowest impurity value, followed by N_2 and N_3. Despite their consistency, the attribute chosen as the test condition may vary depending on the choice of impurity measure, as will be shown in Exercise 3 on page 198.

To determine how well a test condition performs, we need to compare the degree of impurity of the parent node (before splitting) with the degree of impurity of the child nodes (after splitting). The larger their difference, the better the test condition. The gain, Δ, is a criterion that can be used to determine the goodness of a split:

$$\Delta = I(\text{parent}) - \sum_{j=1}^{k} \frac{N(v_j)}{N} I(v_j), \tag{4.6}$$

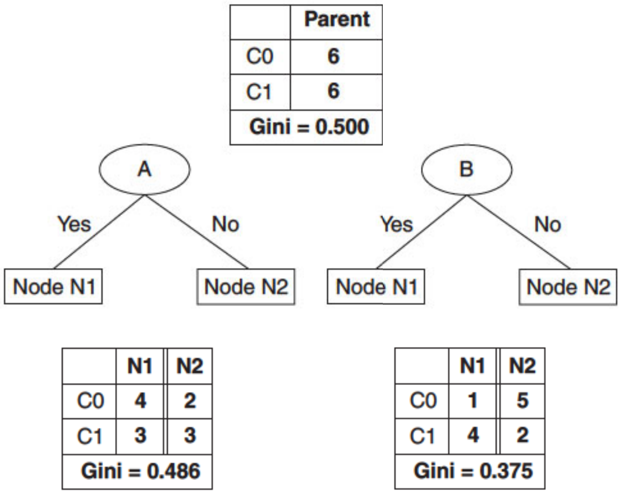
where $I(\cdot)$ is the impurity measure of a given node, N is the total number of records at the parent node, k is the number of attribute values, and $N(v_j)$ is the number of records associated with the child node, v_j. Decision tree induction algorithms often choose a test condition that maximizes the gain Δ. Since $I(\text{parent})$ is the same for all test conditions, maximizing the gain is equivalent to minimizing the weighted average impurity measures of the child nodes. Finally, when entropy is used as the impurity measure in Equation 4.6, the difference in entropy is known as the **information gain**, Δ_{info}.

Splitting of Binary Attributes

Consider the diagram shown in Figure 4.14. Suppose there are two ways to split the data into smaller subsets. Before splitting, the Gini index is 0.5 since there are an equal number of records from both classes. If attribute A is chosen to split the data, the Gini index for node N1 is 0.4898, and for node N2, it is 0.480. The weighted average of the Gini index for the descendent nodes is $(7/12) \times 0.4898 + (5/12) \times 0.480 = 0.486$. Similarly, we can show that the weighted average of the Gini index for attribute B is 0.375. Since the subsets for attribute B have a smaller Gini index, it is preferred over attribute A.

Splitting of Nominal Attributes

As previously noted, a nominal attribute can produce either binary or multiway splits, as shown in Figure 4.15. The computation of the Gini index for a binary split is similar to that shown for determining binary attributes. For the first binary grouping of the `Car Type` attribute, the Gini index of {`Sports,`

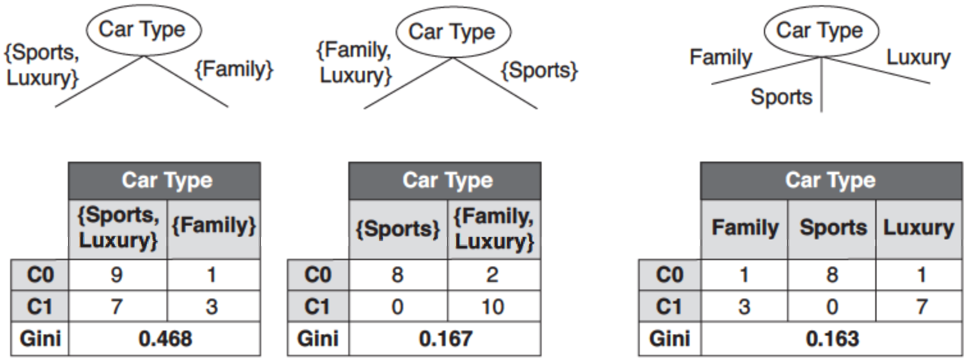


	Parent
C0	6
C1	6
Gini = 0.500	

	N1	N2
C0	4	2
C1	3	3
Gini = 0.486		

	N1	N2
C0	1	5
C1	4	2
Gini = 0.375		

Figure 4.14. Splitting binary attributes.



	Car Type	
	{Sports, Luxury}	{Family}
C0	9	1
C1	7	3
Gini	0.468	

	Car Type	
	{Sports}	{Family, Luxury}
C0	8	2
C1	0	10
Gini	0.167	

(a) Binary split

	Car Type		
	Family	Sports	Luxury
C0	1	8	1
C1	3	0	7
Gini	0.163		

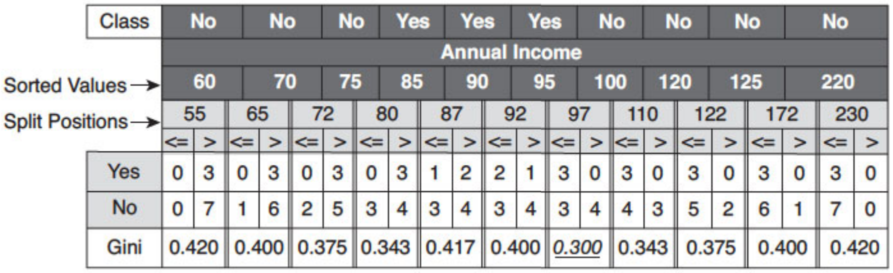
(b) Multiway split

Figure 4.15. Splitting nominal attributes.

Luxury} is 0.4922 and the Gini index of {Family} is 0.3750. The weighted average Gini index for the grouping is equal to

$$16/20 \times 0.4922 + 4/20 \times 0.3750 = 0.468.$$

Similarly, for the second binary grouping of {Sports} and {Family, Luxury}, the weighted average Gini index is 0.167. The second grouping has a lower Gini index because its corresponding subsets are much purer.



Class	No	No	No	Yes	Yes	Yes	No	No	No	No
	Annual Income									
Sorted Values →	60	70	75	85	90	95	100	120	125	220

Split Positions →	55		65		72		80		87		92		97		110		122		172		230	
	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>
Yes	0	3	0	3	0	3	0	3	1	2	2	1	3	0	3	0	3	0	3	0	3	0
No	0	7	1	6	2	5	3	4	3	4	3	4	3	4	4	3	5	2	6	1	7	0
Gini	0.420		0.400		0.375		0.343		0.417		0.400		*0.300*		0.343		0.375		0.400		0.420	

Figure 4.16. Splitting continuous attributes.

For the multiway split, the Gini index is computed for every attribute value. Since Gini({**Family**}) = 0.375, Gini({**Sports**}) = 0, and Gini({**Luxury**}) = 0.219, the overall Gini index for the multiway split is equal to

$$4/20 \times 0.375 + 8/20 \times 0 + 8/20 \times 0.219 = 0.163.$$

The multiway split has a smaller Gini index compared to both two-way splits. This result is not surprising because the two-way split actually merges some of the outcomes of a multiway split, and thus, results in less pure subsets.

Splitting of Continuous Attributes

Consider the example shown in Figure 4.16, in which the test condition **Annual Income** $\leq v$ is used to split the training records for the loan default classification problem. A brute-force method for finding v is to consider every value of the attribute in the N records as a candidate split position. For each candidate v, the data set is scanned once to count the number of records with annual income less than or greater than v. We then compute the Gini index for each candidate and choose the one that gives the lowest value. This approach is computationally expensive because it requires $O(N)$ operations to compute the Gini index at each candidate split position. Since there are N candidates, the overall complexity of this task is $O(N^2)$. To reduce the complexity, the training records are sorted based on their annual income, a computation that requires $O(N \log N)$ time. Candidate split positions are identified by taking the midpoints between two adjacent sorted values: 55, 65, 72, and so on. However, unlike the brute-force approach, we do not have to examine all N records when evaluating the Gini index of a candidate split position.

For the first candidate, $v = 55$, none of the records has annual income less than \$55K. As a result, the Gini index for the descendent node with **Annual**

Income $<$ \$55K is zero. On the other hand, the number of records with annual income greater than or equal to \$55K is 3 (for class **Yes**) and 7 (for class **No**), respectively. Thus, the Gini index for this node is 0.420. The overall Gini index for this candidate split position is equal to $0 \times 0 + 1 \times 0.420 = 0.420$.

For the second candidate, $v = 65$, we can determine its class distribution by updating the distribution of the previous candidate. More specifically, the new distribution is obtained by examining the class label of the record with the lowest annual income (i.e., \$60K). Since the class label for this record is **No**, the count for class **No** is increased from 0 to 1 (for **Annual Income** $\leq$ \$65K) and is decreased from 7 to 6 (for **Annual Income** $>$ \$65K). The distribution for class **Yes** remains unchanged. The new weighted-average Gini index for this candidate split position is 0.400.

This procedure is repeated until the Gini index values for all candidates are computed, as shown in Figure 4.16. The best split position corresponds to the one that produces the smallest Gini index, i.e., $v = 97$. This procedure is less expensive because it requires a constant amount of time to update the class distribution at each candidate split position. It can be further optimized by considering only candidate split positions located between two adjacent records with different class labels. For example, because the first three sorted records (with annual incomes \$60K, \$70K, and \$75K) have identical class labels, the best split position should not reside between \$60K and \$75K. Therefore, the candidate split positions at $v =$ \$55K, \$65K, \$72K, \$87K, \$92K, \$110K, \$122K, \$172K, and \$230K are ignored because they are located between two adjacent records with the same class labels. This approach allows us to reduce the number of candidate split positions from 11 to 2.

Gain Ratio

Impurity measures such as entropy and Gini index tend to favor attributes that have a large number of distinct values. Figure 4.12 shows three alternative test conditions for partitioning the data set given in Exercise 2 on page 198. Comparing the first test condition, **Gender**, with the second, **Car Type**, it is easy to see that **Car Type** seems to provide a better way of splitting the data since it produces purer descendent nodes. However, if we compare both conditions with **Customer ID**, the latter appears to produce purer partitions. Yet **Customer ID** is not a predictive attribute because its value is unique for each record. Even in a less extreme situation, a test condition that results in a large number of outcomes may not be desirable because the number of records associated with each partition is too small to enable us to make any reliable predictions.

There are two strategies for overcoming this problem. The first strategy is to restrict the test conditions to binary splits only. This strategy is employed by decision tree algorithms such as CART. Another strategy is to modify the splitting criterion to take into account the number of outcomes produced by the attribute test condition. For example, in the C4.5 decision tree algorithm, a splitting criterion known as **gain ratio** is used to determine the goodness of a split. This criterion is defined as follows:

$$\text{Gain ratio} = \frac{\Delta_{\text{info}}}{\text{Split Info}}. \tag{4.7}$$

Here, Split Info $= -\sum_{i=1}^{k} P(v_i)\log_2 P(v_i)$ and k is the total number of splits. For example, if each attribute value has the same number of records, then $\forall i : P(v_i) = 1/k$ and the split information would be equal to $\log_2 k$. This example suggests that if an attribute produces a large number of splits, its split information will also be large, which in turn reduces its gain ratio.

4.3.5 Algorithm for Decision Tree Induction

A skeleton decision tree induction algorithm called `TreeGrowth` is shown in Algorithm 4.1. The input to this algorithm consists of the training records E and the attribute set F. The algorithm works by recursively selecting the best attribute to split the data (Step 7) and expanding the leaf nodes of the

Algorithm 4.1 A skeleton decision tree induction algorithm.

```
TreeGrowth ($E$, $F$)
 1: if stopping_cond($E$,$F$) = $true$ then
 2:    $leaf$ = createNode().
 3:    $leaf.label$ = Classify($E$).
 4:    return $leaf$.
 5: else
 6:    $root$ = createNode().
 7:    $root.test\_cond$ = find_best_split($E$, $F$).
 8:    let $V = \{v | v$ is a possible outcome of $root.test\_cond$ $\}$.
 9:    for each $v \in V$ do
10:       $E_v = \{e \mid root.test\_cond(e) = v$ and $e \in E\}$.
11:       $child$ = TreeGrowth($E_v$, $F$).
12:       add $child$ as descendent of $root$ and label the edge $(root \rightarrow child)$ as $v$.
13:    end for
14: end if
15: return $root$.
```

tree (Steps 11 and 12) until the stopping criterion is met (Step 1). The details of this algorithm are explained below:

1. The **createNode**() function extends the decision tree by creating a new node. A node in the decision tree has either a test condition, denoted as *node.test_cond*, or a class label, denoted as *node.label*.

2. The **find_best_split**() function determines which attribute should be selected as the test condition for splitting the training records. As previously noted, the choice of test condition depends on which impurity measure is used to determine the goodness of a split. Some widely used measures include entropy, the Gini index, and the χ^2 statistic.

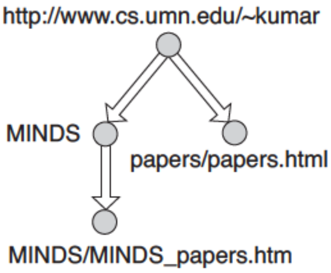
3. The **Classify**() function determines the class label to be assigned to a leaf node. For each leaf node t, let $p(i|t)$ denote the fraction of training records from class i associated with the node t. In most cases, the leaf node is assigned to the class that has the majority number of training records:
$$leaf.label = \operatorname*{argmax}_{i} \; p(i|t), \tag{4.8}$$
where the argmax operator returns the argument i that maximizes the expression $p(i|t)$. Besides providing the information needed to determine the class label of a leaf node, the fraction $p(i|t)$ can also be used to estimate the probability that a record assigned to the leaf node t belongs to class i. Sections 5.7.2 and 5.7.3 describe how such probability estimates can be used to determine the performance of a decision tree under different cost functions.

4. The **stopping_cond**() function is used to terminate the tree-growing process by testing whether all the records have either the same class label or the same attribute values. Another way to terminate the recursive function is to test whether the number of records have fallen below some minimum threshold.

After building the decision tree, a **tree-pruning** step can be performed to reduce the size of the decision tree. Decision trees that are too large are susceptible to a phenomenon known as **overfitting**. Pruning helps by trimming the branches of the initial tree in a way that improves the generalization capability of the decision tree. The issues of overfitting and tree pruning are discussed in more detail in Section 4.4.

Session	IP Address	Timestamp	Request Method	Requested Web Page	Protocol	Status	Number of Bytes	Referrer	User Agent
1	160.11.11.11	08/Aug/2004 10:15:21	GET	http://www.cs.umn.edu/ ~kumar	HTTP/1.1	200	6424		Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:15:34	GET	http://www.cs.umn.edu/ ~kumar/MINDS	HTTP/1.1	200	41378	http://www.cs.umn.edu/ ~kumar	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:15:41	GET	http://www.cs.umn.edu/ ~kumar/MINDS/MINDS _papers.htm	HTTP/1.1	200	1018516	http://www.cs.umn.edu/ ~kumar/MINDS	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:16:11	GET	http://www.cs.umn.edu/ ~kumar/papers/papers. html	HTTP/1.1	200	7463	http://www.cs.umn.edu/ ~kumar	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
2	35.9.2.2	08/Aug/2004 10:16:15	GET	http://www.cs.umn.edu/ ~steinbac	HTTP/1.0	200	3149		Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7) Gecko/20040616

(a) Example of a Web server log.



(b) Graph of a Web session.

Attribute Name	Description
totalPages	Total number of pages retrieved in a Web session
ImagePages	Total number of image pages retrieved in a Web session
TotalTime	Total amount of time spent by Web site visitor
RepeatedAccess	The same page requested more than once in a Web session
ErrorRequest	Errors in requesting for Web pages
GET	Percentage of requests made using GET method
POST	Percentage of requests made using POST method
HEAD	Percentage of requests made using HEAD method
Breadth	Breadth of Web traversal
Depth	Depth of Web traversal
MultiIP	Session with multiple IP addresses
MultiAgent	Session with multiple user agents

(c) Derived attributes for Web robot detection.

Figure 4.17. Input data for Web robot detection.

4.3.6 An Example: Web Robot Detection

Web usage mining is the task of applying data mining techniques to extract useful patterns from Web access logs. These patterns can reveal interesting characteristics of site visitors; e.g., people who repeatedly visit a Web site and view the same product description page are more likely to buy the product if certain incentives such as rebates or free shipping are offered.

In Web usage mining, it is important to distinguish accesses made by human users from those due to Web robots. A Web robot (also known as a Web crawler) is a software program that automatically locates and retrieves information from the Internet by following the hyperlinks embedded in Web pages. These programs are deployed by search engine portals to gather the documents necessary for indexing the Web. Web robot accesses must be discarded before applying Web mining techniques to analyze human browsing behavior.

This section describes how a decision tree classifier can be used to distinguish between accesses by human users and those by Web robots. The input data was obtained from a Web server log, a sample of which is shown in Figure 4.17(a). Each line corresponds to a single page request made by a Web client (a user or a Web robot). The fields recorded in the Web log include the IP address of the client, timestamp of the request, Web address of the requested document, size of the document, and the client's identity (via the user agent field). A Web session is a sequence of requests made by a client during a single visit to a Web site. Each Web session can be modeled as a directed graph, in which the nodes correspond to Web pages and the edges correspond to hyperlinks connecting one Web page to another. Figure 4.17(b) shows a graphical representation of the first Web session given in the Web server log.

To classify the Web sessions, features are constructed to describe the characteristics of each session. Figure 4.17(c) shows some of the features used for the Web robot detection task. Among the notable features include the **`depth`** and **`breadth`** of the traversal. **`Depth`** determines the maximum distance of a requested page, where distance is measured in terms of the number of hyperlinks away from the entry point of the Web site. For example, the home page **`http://www.cs.umn.edu/∼kumar`** is assumed to be at depth 0, whereas **`http://www.cs.umn.edu/kumar/MINDS/MINDS_papers.htm`** is located at depth 2. Based on the Web graph shown in Figure 4.17(b), the **`depth`** attribute for the first session is equal to two. The **`breadth`** attribute measures the width of the corresponding Web graph. For example, the **`breadth`** of the Web session shown in Figure 4.17(b) is equal to two.

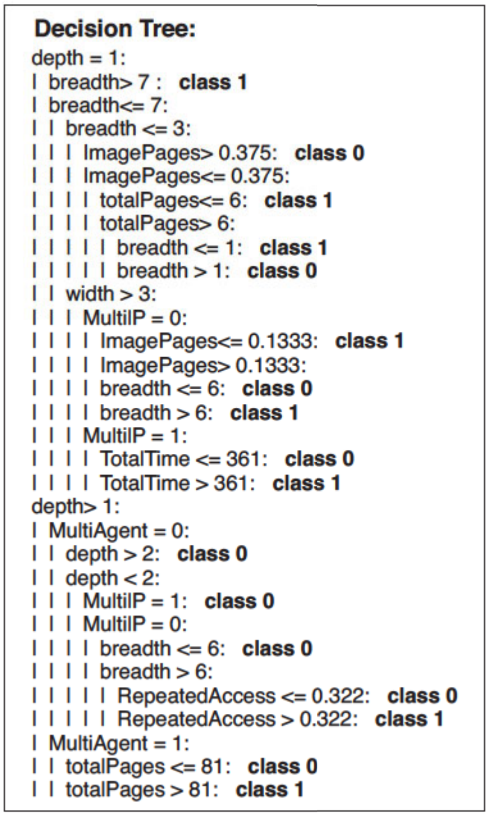
The data set for classification contains 2916 records, with equal numbers of sessions due to Web robots (class 1) and human users (class 0). 10% of the data were reserved for training while the remaining 90% were used for testing. The induced decision tree model is shown in Figure 4.18. The tree has an error rate equal to 3.8% on the training set and 5.3% on the test set.

The model suggests that Web robots can be distinguished from human users in the following way:

1. Accesses by Web robots tend to be broad but shallow, whereas accesses by human users tend to be more focused (narrow but deep).

2. Unlike human users, Web robots seldom retrieve the image pages associated with a Web document.

3. Sessions due to Web robots tend to be long and contain a large number of requested pages.



```
Decision Tree:
depth = 1:
| breadth> 7 :  class 1
| breadth<= 7:
| | breadth <= 3:
| | | ImagePages> 0.375:  class 0
| | | ImagePages<= 0.375:
| | | | totalPages<= 6:  class 1
| | | | totalPages> 6:
| | | | | breadth <= 1:  class 1
| | | | | breadth > 1:  class 0
| | width > 3:
| | | MultiIP = 0:
| | | | ImagePages<= 0.1333:  class 1
| | | | ImagePages> 0.1333:
| | | | breadth <= 6:  class 0
| | | | breadth > 6:  class 1
| | | MultiIP = 1:
| | | | TotalTime <= 361:  class 0
| | | | TotalTime > 361:  class 1
depth> 1:
| MultiAgent = 0:
| | depth > 2:  class 0
| | depth < 2:
| | | MultiIP = 1:  class 0
| | | MultiIP = 0:
| | | | breadth <= 6:  class 0
| | | | breadth > 6:
| | | | | RepeatedAccess <= 0.322:  class 0
| | | | | RepeatedAccess > 0.322:  class 1
| MultiAgent = 1:
| | totalPages <= 81:  class 0
| | totalPages > 81:  class 1
```

Figure 4.18. Decision tree model for Web robot detection.

4. Web robots are more likely to make repeated requests for the same document since the Web pages retrieved by human users are often cached by the browser.

4.3.7 Characteristics of Decision Tree Induction

The following is a summary of the important characteristics of decision tree induction algorithms.

1. Decision tree induction is a nonparametric approach for building classification models. In other words, it does not require any prior assumptions regarding the type of probability distributions satisfied by the class and other attributes (unlike some of the techniques described in Chapter 5).

2. Finding an optimal decision tree is an NP-complete problem. Many decision tree algorithms employ a heuristic-based approach to guide their search in the vast hypothesis space. For example, the algorithm presented in Section 4.3.5 uses a greedy, top-down, recursive partitioning strategy for growing a decision tree.

3. Techniques developed for constructing decision trees are computationally inexpensive, making it possible to quickly construct models even when the training set size is very large. Furthermore, once a decision tree has been built, classifying a test record is extremely fast, with a worst-case complexity of $O(w)$, where w is the maximum depth of the tree.

4. Decision trees, especially smaller-sized trees, are relatively easy to interpret. The accuracies of the trees are also comparable to other classification techniques for many simple data sets.

5. Decision trees provide an expressive representation for learning discrete-valued functions. However, they do not generalize well to certain types of Boolean problems. One notable example is the parity function, whose value is 0 (1) when there is an odd (even) number of Boolean attributes with the value $True$. Accurate modeling of such a function requires a full decision tree with 2^d nodes, where d is the number of Boolean attributes (see Exercise 1 on page 198).

6. Decision tree algorithms are quite robust to the presence of noise, especially when methods for avoiding overfitting, as described in Section 4.4, are employed.

7. The presence of redundant attributes does not adversely affect the accuracy of decision trees. An attribute is redundant if it is strongly correlated with another attribute in the data. One of the two redundant attributes will not be used for splitting once the other attribute has been chosen. However, if the data set contains many irrelevant attributes, i.e., attributes that are not useful for the classification task, then some of the irrelevant attributes may be accidently chosen during the tree-growing process, which results in a decision tree that is larger than necessary. Feature selection techniques can help to improve the accuracy of decision trees by eliminating the irrelevant attributes during preprocessing. We will investigate the issue of too many irrelevant attributes in Section 4.4.3.

8. Since most decision tree algorithms employ a top-down, recursive partitioning approach, the number of records becomes smaller as we traverse down the tree. At the leaf nodes, the number of records may be too small to make a statistically significant decision about the class representation of the nodes. This is known as the **data fragmentation** problem. One possible solution is to disallow further splitting when the number of records falls below a certain threshold.

9. A subtree can be replicated multiple times in a decision tree, as illustrated in Figure 4.19. This makes the decision tree more complex than necessary and perhaps more difficult to interpret. Such a situation can arise from decision tree implementations that rely on a single attribute test condition at each internal node. Since most of the decision tree algorithms use a divide-and-conquer partitioning strategy, the same test condition can be applied to different parts of the attribute space, thus leading to the subtree replication problem.

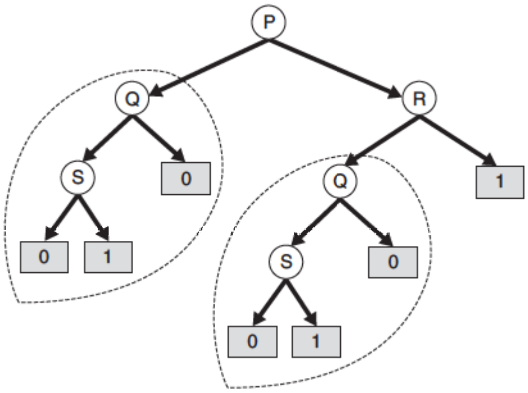


Figure 4.19. Tree replication problem. The same subtree can appear at different branches.

10. The test conditions described so far in this chapter involve using only a single attribute at a time. As a consequence, the tree-growing procedure can be viewed as the process of partitioning the attribute space into disjoint regions until each region contains records of the same class (see Figure 4.20). The border between two neighboring regions of different classes is known as a **decision boundary**. Since the test condition involves only a single attribute, the decision boundaries are rectilinear; i.e., parallel to the "coordinate axes." This limits the expressiveness of the

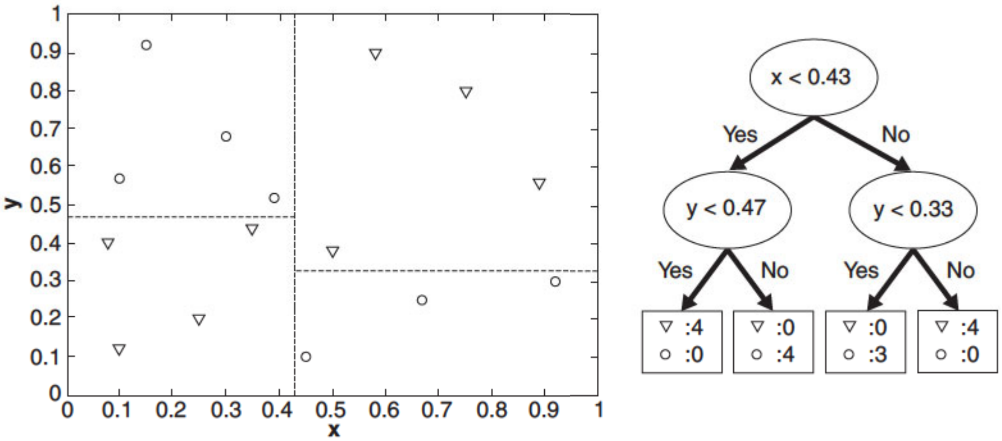


Figure 4.20. Example of a decision tree and its decision boundaries for a two-dimensional data set.

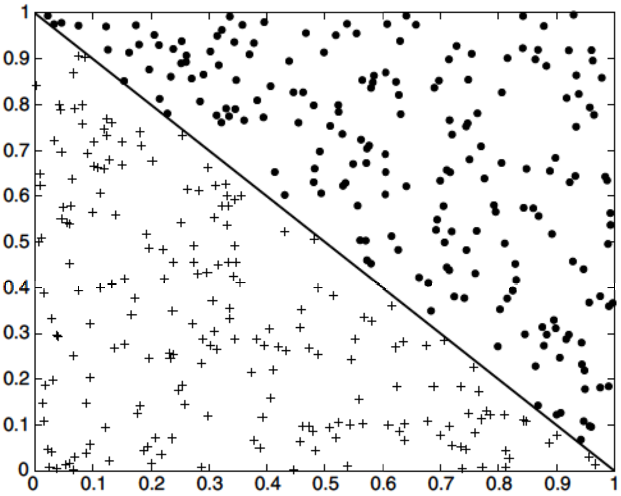


Figure 4.21. Example of data set that cannot be partitioned optimally using test conditions involving single attributes.

decision tree representation for modeling complex relationships among continuous attributes. Figure 4.21 illustrates a data set that cannot be classified effectively by a decision tree algorithm that uses test conditions involving only a single attribute at a time.

An **oblique decision tree** can be used to overcome this limitation because it allows test conditions that involve more than one attribute. The data set given in Figure 4.21 can be easily represented by an oblique decision tree containing a single node with test condition

$$x + y < 1.$$

Although such techniques are more expressive and can produce more compact trees, finding the optimal test condition for a given node can be computationally expensive.

Constructive induction provides another way to partition the data into homogeneous, nonrectangular regions (see Section 2.3.5 on page 57). This approach creates composite attributes representing an arithmetic or logical combination of the existing attributes. The new attributes provide a better discrimination of the classes and are augmented to the data set prior to decision tree induction. Unlike the oblique decision tree approach, constructive induction is less expensive because it identifies all the relevant combinations of attributes once, prior to constructing the decision tree. In contrast, an oblique decision tree must determine the right attribute combination dynamically, every time an internal node is expanded. However, constructive induction can introduce attribute redundancy in the data since the new attribute is a combination of several existing attributes.

11. Studies have shown that the choice of impurity measure has little effect on the performance of decision tree induction algorithms. This is because many impurity measures are quite consistent with each other, as shown in Figure 4.13 on page 159. Indeed, the strategy used to prune the tree has a greater impact on the final tree than the choice of impurity measure.

4.4 Model Overfitting

The errors committed by a classification model are generally divided into two types: **training errors** and **generalization errors**. Training error, also known as **resubstitution error** or **apparent error**, is the number of misclassification errors committed on training records, whereas generalization error is the expected error of the model on previously unseen records.

Recall from Section 4.2 that a good classification model must not only fit the training data well, it must also accurately classify records it has never

An **oblique decision tree** can be used to overcome this limitation because it allows test conditions that involve more than one attribute. The data set given in Figure 4.21 can be easily represented by an oblique decision tree containing a single node with test condition

$$x + y < 1.$$

Although such techniques are more expressive and can produce more compact trees, finding the optimal test condition for a given node can be computationally expensive.

Constructive induction provides another way to partition the data into homogeneous, nonrectangular regions (see Section 2.3.5 on page 57). This approach creates composite attributes representing an arithmetic or logical combination of the existing attributes. The new attributes provide a better discrimination of the classes and are augmented to the data set prior to decision tree induction. Unlike the oblique decision tree approach, constructive induction is less expensive because it identifies all the relevant combinations of attributes once, prior to constructing the decision tree. In contrast, an oblique decision tree must determine the right attribute combination dynamically, every time an internal node is expanded. However, constructive induction can introduce attribute redundancy in the data since the new attribute is a combination of several existing attributes.

11. Studies have shown that the choice of impurity measure has little effect on the performance of decision tree induction algorithms. This is because many impurity measures are quite consistent with each other, as shown in Figure 4.13 on page 159. Indeed, the strategy used to prune the tree has a greater impact on the final tree than the choice of impurity measure.

4.4 Model Overfitting

The errors committed by a classification model are generally divided into two types: **training errors** and **generalization errors**. Training error, also known as **resubstitution error** or **apparent error**, is the number of misclassification errors committed on training records, whereas generalization error is the expected error of the model on previously unseen records.

Recall from Section 4.2 that a good classification model must not only fit the training data well, it must also accurately classify records it has never

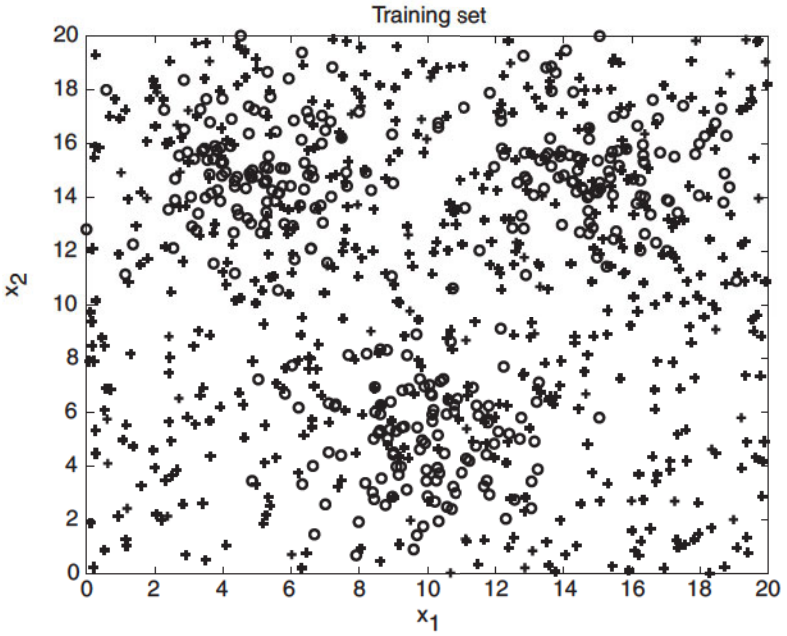


Figure 4.22. Example of a data set with binary classes.

seen before. In other words, a good model must have low training error as well as low generalization error. This is important because a model that fits the training data too well can have a poorer generalization error than a model with a higher training error. Such a situation is known as model overfitting.

Overfitting Example in Two-Dimensional Data For a more concrete example of the overfitting problem, consider the two-dimensional data set shown in Figure 4.22. The data set contains data points that belong to two different classes, denoted as class o and class +, respectively. The data points for the o class are generated from a mixture of three Gaussian distributions, while a uniform distribution is used to generate the data points for the + class. There are altogether 1200 points belonging to the o class and 1800 points belonging to the + class. 30% of the points are chosen for training, while the remaining 70% are used for testing. A decision tree classifier that uses the Gini index as its impurity measure is then applied to the training set. To investigate the effect of overfitting, different levels of pruning are applied to the initial, fully-grown tree. Figure 4.23(b) shows the training and test error rates of the decision tree.

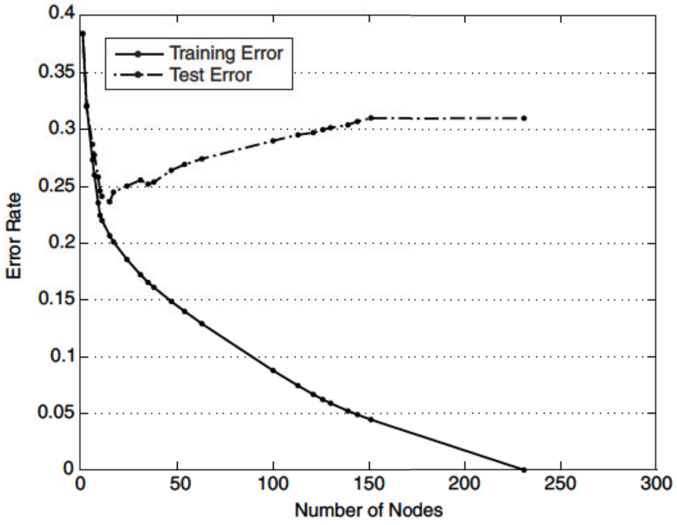
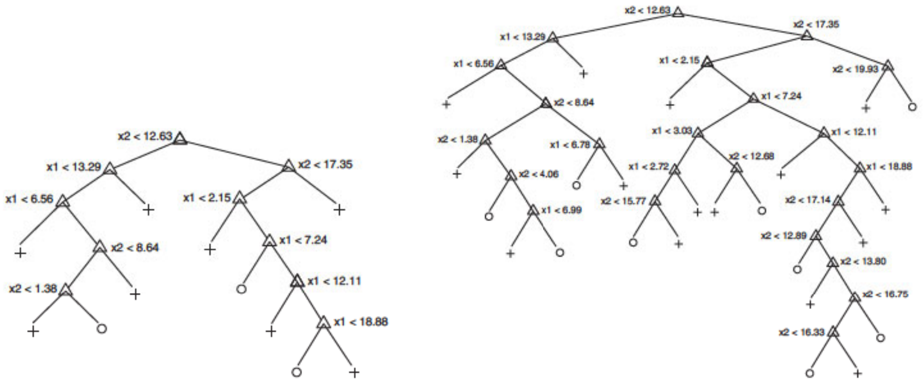


Figure 4.23. Training and test error rates.

Notice that the training and test error rates of the model are large when the size of the tree is very small. This situation is known as **model underfitting**. Underfitting occurs because the model has yet to learn the true structure of the data. As a result, it performs poorly on both the training and the test sets. As the number of nodes in the decision tree increases, the tree will have fewer training and test errors. However, once the tree becomes too large, its test error rate begins to increase even though its training error rate continues to decrease. This phenomenon is known as **model overfitting**.

To understand the overfitting phenomenon, note that the training error of a model can be reduced by increasing the model complexity. For example, the leaf nodes of the tree can be expanded until it perfectly fits the training data. Although the training error for such a complex tree is zero, the test error can be large because the tree may contain nodes that accidently fit some of the noise points in the training data. Such nodes can degrade the performance of the tree because they do not generalize well to the test examples. Figure 4.24 shows the structure of two decision trees with different number of nodes. The tree that contains the smaller number of nodes has a higher training error rate, but a lower test error rate compared to the more complex tree.

Overfitting and underfitting are two pathologies that are related to the model complexity. The remainder of this section examines some of the potential causes of model overfitting.



(a) Decision tree with 11 leaf nodes.

(b) Decision tree with 24 leaf nodes.

Figure 4.24. Decision trees with different model complexities.

4.4.1 Overfitting Due to Presence of Noise

Consider the training and test sets shown in Tables 4.3 and 4.4 for the mammal classification problem. Two of the ten training records are mislabeled: bats and whales are classified as non-mammals instead of mammals.

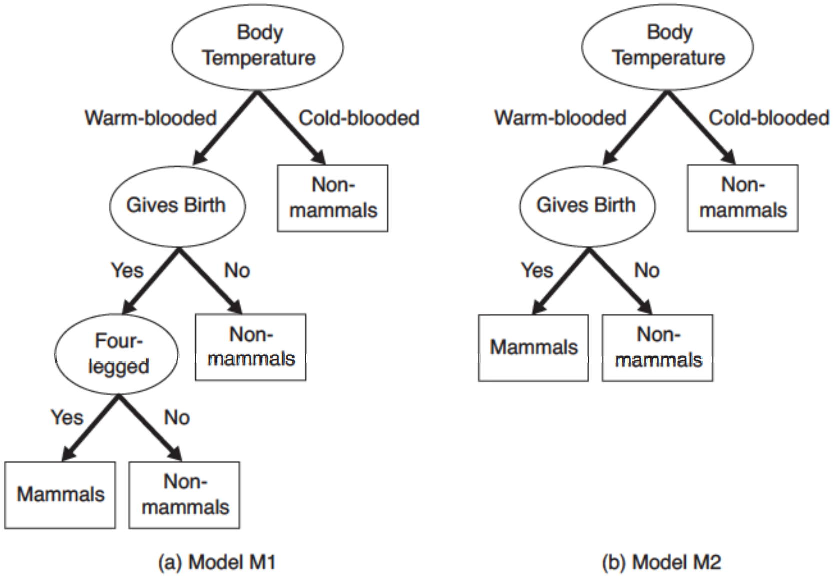
A decision tree that perfectly fits the training data is shown in Figure 4.25(a). Although the training error for the tree is zero, its error rate on

Table 4.3. An example training set for classifying mammals. Class labels with asterisk symbols represent mislabeled records.

Name	Body Temperature	Gives Birth	Four-legged	Hibernates	Class Label
porcupine	warm-blooded	yes	yes	yes	yes
cat	warm-blooded	yes	yes	no	yes
bat	warm-blooded	yes	no	yes	no*
whale	warm-blooded	yes	no	no	no*
salamander	cold-blooded	no	yes	yes	no
komodo dragon	cold-blooded	no	yes	no	no
python	cold-blooded	no	no	yes	no
salmon	cold-blooded	no	no	no	no
eagle	warm-blooded	no	no	no	no
guppy	cold-blooded	yes	no	no	no

Table 4.4. An example test set for classifying mammals.

Name	Body Temperature	Gives Birth	Four-legged	Hibernates	Class Label
human	warm-blooded	yes	no	no	yes
pigeon	warm-blooded	no	no	no	no
elephant	warm-blooded	yes	yes	no	yes
leopard shark	cold-blooded	yes	no	no	no
turtle	cold-blooded	no	yes	no	no
penguin	cold-blooded	no	no	no	no
eel	cold-blooded	no	no	no	no
dolphin	warm-blooded	yes	no	no	yes
spiny anteater	warm-blooded	no	yes	yes	yes
gila monster	cold-blooded	no	yes	yes	no



(a) Model M1

(b) Model M2

Figure 4.25. Decision tree induced from the data set shown in Table 4.3.

the test set is 30%. Both humans and dolphins were misclassified as non-mammals because their attribute values for `Body Temperature`, `Gives Birth`, and `Four-legged` are identical to the mislabeled records in the training set. Spiny anteaters, on the other hand, represent an exceptional case in which the class label of a test record contradicts the class labels of other similar records in the training set. Errors due to exceptional cases are often unavoidable and establish the minimum error rate achievable by any classifier.

In contrast, the decision tree $M2$ shown in Figure 4.25(b) has a lower test error rate (10%) even though its training error rate is somewhat higher (20%). It is evident that the first decision tree, $M1$, has overfitted the training data because there is a simpler model with lower error rate on the test set. The `Four-legged` attribute test condition in model $M1$ is spurious because it fits the mislabeled training records, which leads to the misclassification of records in the test set.

4.4.2 Overfitting Due to Lack of Representative Samples

Models that make their classification decisions based on a small number of training records are also susceptible to overfitting. Such models can be generated because of lack of representative samples in the training data and learning algorithms that continue to refine their models even when few training records are available. We illustrate these effects in the example below.

Consider the five training records shown in Table 4.5. All of these training records are labeled correctly and the corresponding decision tree is depicted in Figure 4.26. Although its training error is zero, its error rate on the test set is 30%.

Table 4.5. An example training set for classifying mammals.

Name	Body Temperature	Gives Birth	Four-legged	Hibernates	Class Label
salamander	cold-blooded	no	yes	yes	no
guppy	cold-blooded	yes	no	no	no
eagle	warm-blooded	no	no	no	no
poorwill	warm-blooded	no	no	yes	no
platypus	warm-blooded	no	yes	yes	yes

Humans, elephants, and dolphins are misclassified because the decision tree classifies all warm-blooded vertebrates that do not hibernate as non-mammals. The tree arrives at this classification decision because there is only one training record, which is an eagle, with such characteristics. This example clearly demonstrates the danger of making wrong predictions when there are not enough representative examples at the leaf nodes of a decision tree.

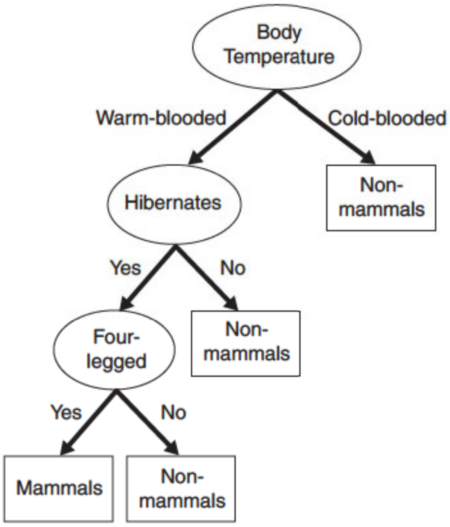


Figure 4.26. Decision tree induced from the data set shown in Table 4.5.

4.4.3 Overfitting and the Multiple Comparison Procedure

Model overfitting may arise in learning algorithms that employ a methodology known as multiple comparison procedure. To understand multiple comparison procedure, consider the task of predicting whether the stock market will rise or fall in the next ten trading days. If a stock analyst simply makes random guesses, the probability that her prediction is correct on any trading day is 0.5. However, the probability that she will predict correctly at least eight out of the ten times is

$$\frac{\binom{10}{8} + \binom{10}{9} + \binom{10}{10}}{2^{10}} = 0.0547,$$

which seems quite unlikely.

Suppose we are interested in choosing an investment advisor from a pool of fifty stock analysts. Our strategy is to select the analyst who makes the most correct predictions in the next ten trading days. The flaw in this strategy is that even if all the analysts had made their predictions in a random fashion, the probability that at least one of them makes at least eight correct predictions is

$$1 - (1 - 0.0547)^{50} = 0.9399,$$

which is very high. Although each analyst has a low probability of predicting at least eight times correctly, putting them together, we have a high probability of finding an analyst who can do so. Furthermore, there is no guarantee in the future that such an analyst will continue to make accurate predictions through random guessing.

How does the multiple comparison procedure relate to model overfitting? Many learning algorithms explore a set of independent alternatives, $\{\gamma_i\}$, and then choose an alternative, $\gamma_{\max}$, that maximizes a given criterion function. The algorithm will add $\gamma_{\max}$ to the current model in order to improve its overall performance. This procedure is repeated until no further improvement is observed. As an example, during decision tree growing, multiple tests are performed to determine which attribute can best split the training data. The attribute that leads to the best split is chosen to extend the tree as long as the observed improvement is statistically significant.

Let T_0 be the initial decision tree and T_x be the new tree after inserting an internal node for attribute x. In principle, x can be added to the tree if the observed gain, $\Delta(T_0, T_x)$, is greater than some predefined threshold α. If there is only one attribute test condition to be evaluated, then we can avoid inserting spurious nodes by choosing a large enough value of α. However, in practice, more than one test condition is available and the decision tree algorithm must choose the best attribute $x_{\max}$ from a set of candidates, $\{x_1, x_2, \ldots, x_k\}$, to partition the data. In this situation, the algorithm is actually using a multiple comparison procedure to decide whether a decision tree should be extended. More specifically, it is testing for $\Delta(T_0, T_{x_{\max}}) > \alpha$ instead of $\Delta(T_0, T_x) > \alpha$. As the number of alternatives, k, increases, so does our chance of finding $\Delta(T_0, T_{x_{\max}}) > \alpha$. Unless the gain function Δ or threshold α is modified to account for k, the algorithm may inadvertently add spurious nodes to the model, which leads to model overfitting.

This effect becomes more pronounced when the number of training records from which $x_{\max}$ is chosen is small, because the variance of $\Delta(T_0, T_{x_{\max}})$ is high when fewer examples are available for training. As a result, the probability of finding $\Delta(T_0, T_{x_{\max}}) > \alpha$ increases when there are very few training records. This often happens when the decision tree grows deeper, which in turn reduces the number of records covered by the nodes and increases the likelihood of adding unnecessary nodes into the tree. Failure to compensate for the large number of alternatives or the small number of training records will therefore lead to model overfitting.

4.4.4 Estimation of Generalization Errors

Although the primary reason for overfitting is still a subject of debate, it is generally agreed that the complexity of a model has an impact on model overfitting, as was illustrated in Figure 4.23. The question is, how do we determine the right model complexity? The ideal complexity is that of a model that produces the lowest generalization error. The problem is that the learning algorithm has access only to the training set during model building (see Figure 4.3). It has no knowledge of the test set, and thus, does not know how well the tree will perform on records it has never seen before. The best it can do is to estimate the generalization error of the induced tree. This section presents several methods for doing the estimation.

Using Resubstitution Estimate

The resubstitution estimate approach assumes that the training set is a good representation of the overall data. Consequently, the training error, otherwise known as resubstitution error, can be used to provide an optimistic estimate for the generalization error. Under this assumption, a decision tree induction algorithm simply selects the model that produces the lowest training error rate as its final model. However, the training error is usually a poor estimate of generalization error.

Example 4.1. Consider the binary decision trees shown in Figure 4.27. Assume that both trees are generated from the same training data and both make their classification decisions at each leaf node according to the majority class. Note that the left tree, T_L, is more complex because it expands some of the leaf nodes in the right tree, T_R. The training error rate for the left tree is $e(T_L) = 4/24 = 0.167$, while the training error rate for the right tree is

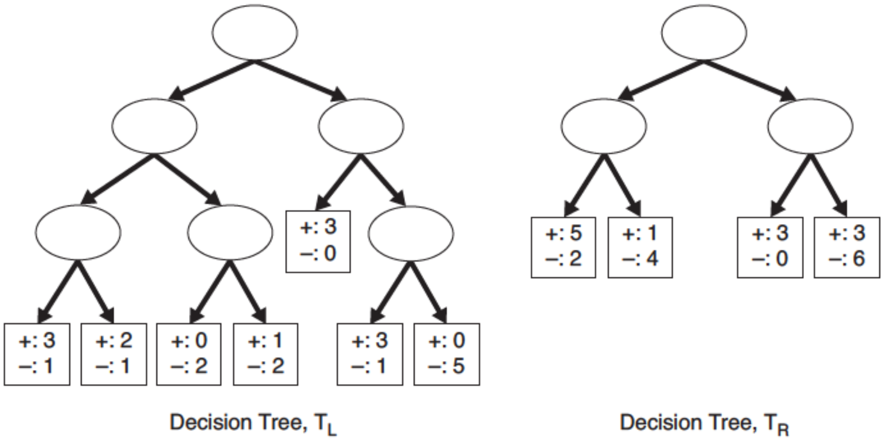


Figure 4.27. Example of two decision trees generated from the same training data.

$e(T_R) = 6/24 = 0.25$. Based on their resubstitution estimate, the left tree is considered better than the right tree. ■

Incorporating Model Complexity

As previously noted, the chance for model overfitting increases as the model becomes more complex. For this reason, we should prefer simpler models, a strategy that agrees with a well-known principle known as **Occam's razor** or the **principle of parsimony**:

Definition 4.2. Occam's Razor: Given two models with the same generalization errors, the simpler model is preferred over the more complex model.

Occam's razor is intuitive because the additional components in a complex model stand a greater chance of being fitted purely by chance. In the words of Einstein, "Everything should be made as simple as possible, but not simpler." Next, we present two methods for incorporating model complexity into the evaluation of classification models.

Pessimistic Error Estimate The first approach explicitly computes generalization error as the sum of training error and a penalty term for model complexity. The resulting generalization error can be considered its pessimistic error estimate. For instance, let $n(t)$ be the number of training records classified by node t and $e(t)$ be the number of misclassified records. The pessimistic error estimate of a decision tree T, $e_g(T)$, can be computed as follows:

$$e_g(T) = \frac{\sum_{i=1}^{k}[e(t_i) + \Omega(t_i)]}{\sum_{i=1}^{k} n(t_i)} = \frac{e(T) + \Omega(T)}{N_t},$$

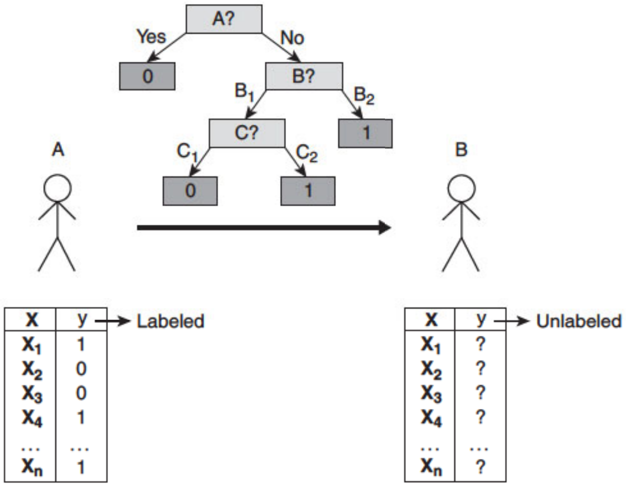
where k is the number of leaf nodes, $e(T)$ is the overall training error of the decision tree, N_t is the number of training records, and $\Omega(t_i)$ is the penalty term associated with each node t_i.

Example 4.2. Consider the binary decision trees shown in Figure 4.27. If the penalty term is equal to 0.5, then the pessimistic error estimate for the left tree is

$$e_g(T_L) = \frac{4 + 7 \times 0.5}{24} = \frac{7.5}{24} = 0.3125$$

and the pessimistic error estimate for the right tree is

$$e_g(T_R) = \frac{6 + 4 \times 0.5}{24} = \frac{8}{24} = 0.3333.$$



X	y
X_1	1
X_2	0
X_3	0
X_4	1
...	...
X_n	1

X	y
X_1	?
X_2	?
X_3	?
X_4	?
...	...
X_n	?

Figure 4.28. The minimum description length (MDL) principle.

Thus, the left tree has a better pessimistic error rate than the right tree. For binary trees, a penalty term of 0.5 means a node should always be expanded into its two child nodes as long as it improves the classification of at least one training record because expanding a node, which is equivalent to adding 0.5 to the overall error, is less costly than committing one training error.

If $\Omega(t) = 1$ for all the nodes t, the pessimistic error estimate for the left tree is $e_g(T_L) = 11/24 = 0.458$, while the pessimistic error estimate for the right tree is $e_g(T_R) = 10/24 = 0.417$. The right tree therefore has a better pessimistic error rate than the left tree. Thus, a node should not be expanded into its child nodes unless it reduces the misclassification error for more than one training record. ■

Minimum Description Length Principle Another way to incorporate model complexity is based on an information-theoretic approach known as the minimum description length or MDL principle. To illustrate this principle, consider the example shown in Figure 4.28. In this example, both A and B are given a set of records with known attribute values $\mathbf{x}$. In addition, person A knows the exact class label for each record, while person B knows none of this information. B can obtain the classification of each record by requesting that A transmits the class labels sequentially. Such a message would require $\Theta(n)$ bits of information, where n is the total number of records.

Alternatively, A may decide to build a classification model that summarizes the relationship between $\mathbf{x}$ and y. The model can be encoded in a compact form before being transmitted to `B`. If the model is 100% accurate, then the cost of transmission is equivalent to the cost of encoding the model. Otherwise, `A` must also transmit information about which record is classified incorrectly by the model. Thus, the overall cost of transmission is

$$Cost(model, data) = Cost(model) + Cost(data|model), \tag{4.9}$$

where the first term on the right-hand side is the cost of encoding the model, while the second term represents the cost of encoding the mislabeled records. According to the MDL principle, we should seek a model that minimizes the overall cost function. An example showing how to compute the total description length of a decision tree is given by Exercise 9 on page 202.

Estimating Statistical Bounds

The generalization error can also be estimated as a statistical correction to the training error. Since generalization error tends to be larger than training error, the statistical correction is usually computed as an upper bound to the training error, taking into account the number of training records that reach a particular leaf node. For instance, in the C4.5 decision tree algorithm, the number of errors committed by each leaf node is assumed to follow a binomial distribution. To compute its generalization error, we must determine the upper bound limit to the observed training error, as illustrated in the next example.

Example 4.3. Consider the left-most branch of the binary decision trees shown in Figure 4.27. Observe that the left-most leaf node of T_R has been expanded into two child nodes in T_L. Before splitting, the error rate of the node is $2/7 = 0.286$. By approximating a binomial distribution with a normal distribution, the following upper bound of the error rate e can be derived:

$$e_{upper}(N, e, \alpha) = \frac{e + \frac{z^2_{\alpha/2}}{2N} + z_{\alpha/2}\sqrt{\frac{e(1-e)}{N} + \frac{z^2_{\alpha/2}}{4N^2}}}{1 + \frac{z^2_{\alpha/2}}{N}}, \tag{4.10}$$

where α is the confidence level, $z_{\alpha/2}$ is the standardized value from a standard normal distribution, and N is the total number of training records used to compute e. By replacing $\alpha = 25\%$, $N = 7$, and $e = 2/7$, the upper bound for the error rate is $e_{upper}(7, 2/7, 0.25) = 0.503$, which corresponds to $7 \times 0.503 = 3.521$ errors. If we expand the node into its child nodes as shown in T_L, the training error rates for the child nodes are $1/4 = 0.250$ and $1/3 = 0.333$, respectively. Using Equation 4.10, the upper bounds of these error rates are $e_{upper}(4, 1/4, 0.25) = 0.537$ and $e_{upper}(3, 1/3, 0.25) = 0.650$, respectively. The overall training error of the child nodes is $4 \times 0.537 + 3 \times 0.650 = 4.098$, which is larger than the estimated error for the corresponding node in T_R. ■

Using a Validation Set

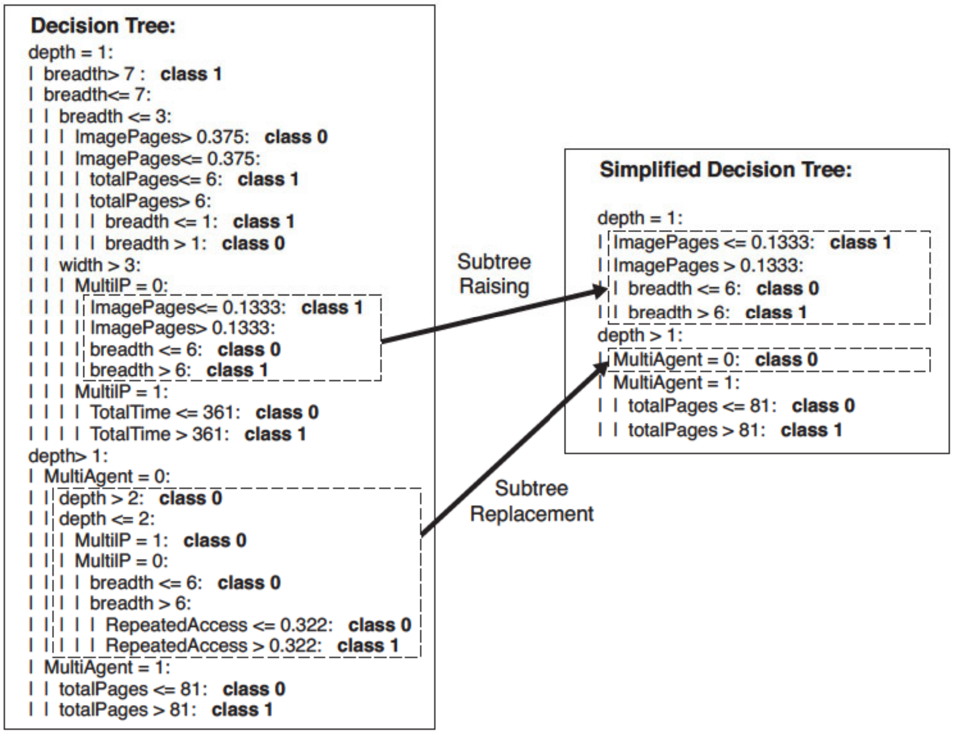
In this approach, instead of using the training set to estimate the generalization error, the original training data is divided into two smaller subsets. One of the subsets is used for training, while the other, known as the validation set, is used for estimating the generalization error. Typically, two-thirds of the training set is reserved for model building, while the remaining one-third is used for error estimation.

This approach is typically used with classification techniques that can be parameterized to obtain models with different levels of complexity. The complexity of the best model can be estimated by adjusting the parameter of the learning algorithm (e.g., the pruning level of a decision tree) until the empirical model produced by the learning algorithm attains the lowest error rate on the validation set. Although this approach provides a better way for estimating how well the model performs on previously unseen records, less data is available for training.

4.4.5 Handling Overfitting in Decision Tree Induction

In the previous section, we described several methods for estimating the generalization error of a classification model. Having a reliable estimate of generalization error allows the learning algorithm to search for an accurate model without overfitting the training data. This section presents two strategies for avoiding model overfitting in the context of decision tree induction.

Prepruning (Early Stopping Rule) In this approach, the tree-growing algorithm is halted before generating a fully grown tree that perfectly fits the entire training data. To do this, a more restrictive stopping condition must be used; e.g., stop expanding a leaf node when the observed gain in impurity measure (or improvement in the estimated generalization error) falls below a certain threshold. The advantage of this approach is that it avoids generating overly complex subtrees that overfit the training data. Nevertheless, it is difficult to choose the right threshold for early termination. Too high of a threshold will result in underfitted models, while a threshold that is set too low may not be sufficient to overcome the model overfitting problem. Furthermore,



Decision Tree:

```
depth = 1:
| breadth> 7 :  class 1
| breadth<= 7:
| | breadth <= 3:
| | | ImagePages> 0.375:  class 0
| | | ImagePages<= 0.375:
| | | | totalPages<= 6:  class 1
| | | | totalPages> 6:
| | | | | breadth <= 1:  class 1
| | | | | breadth > 1:  class 0
| | width > 3:
| | | MultiIP = 0:
| | | | ImagePages<= 0.1333:  class 1
| | | | ImagePages> 0.1333:
| | | | | breadth <= 6:  class 0
| | | | | breadth > 6:  class 1
| | | MultiIP = 1:
| | | | TotalTime <= 361:  class 0
| | | | TotalTime > 361:  class 1
depth> 1:
| MultiAgent = 0:
| | depth > 2:  class 0
| | depth <= 2:
| | | MultiIP = 1:  class 0
| | | MultiIP = 0:
| | | | breadth <= 6:  class 0
| | | | breadth > 6:
| | | | | RepeatedAccess <= 0.322:  class 0
| | | | | RepeatedAccess > 0.322:  class 1
| MultiAgent = 1:
| | totalPages <= 81:  class 0
| | totalPages > 81:  class 1
```

Simplified Decision Tree:

```
depth = 1:
| ImagePages <= 0.1333:  class 1
| ImagePages > 0.1333:
| | breadth <= 6:  class 0
| | breadth > 6:  class 1
depth > 1:
| MultiAgent = 0:  class 0
| MultiAgent = 1:
| | totalPages <= 81:  class 0
| | totalPages > 81:  class 1
```

Figure 4.29. Post-pruning of the decision tree for Web robot detection.

even if no significant gain is obtained using one of the existing attribute test conditions, subsequent splitting may result in better subtrees.

Post-pruning In this approach, the decision tree is initially grown to its maximum size. This is followed by a tree-pruning step, which proceeds to trim the fully grown tree in a bottom-up fashion. Trimming can be done by replacing a subtree with (1) a new leaf node whose class label is determined from the majority class of records affiliated with the subtree, or (2) the most frequently used branch of the subtree. The tree-pruning step terminates when no further improvement is observed. Post-pruning tends to give better results than prepruning because it makes pruning decisions based on a fully grown tree, unlike prepruning, which can suffer from premature termination of the tree-growing process. However, for post-pruning, the additional computations needed to grow the full tree may be wasted when the subtree is pruned.

Figure 4.29 illustrates the simplified decision tree model for the Web robot detection example given in Section 4.3.6. Notice that the subtrees rooted at

depth = 1 have been replaced by one of the branches involving the attribute **ImagePages**. This approach is also known as **subtree raising**. The **depth** > 1 and **MultiAgent** = 0 subtree has been replaced by a leaf node assigned to class 0. This approach is known as **subtree replacement**. The subtree for **depth** > 1 and **MultiAgent** = 1 remains intact.

4.5 Evaluating the Performance of a Classifier

Section 4.4.4 described several methods for estimating the generalization error of a model during training. The estimated error helps the learning algorithm to do **model selection**; i.e., to find a model of the right complexity that is not susceptible to overfitting. Once the model has been constructed, it can be applied to the test set to predict the class labels of previously unseen records.

It is often useful to measure the performance of the model on the test set because such a measure provides an unbiased estimate of its generalization error. The accuracy or error rate computed from the test set can also be used to compare the relative performance of different classifiers on the same domain. However, in order to do this, the class labels of the test records must be known. This section reviews some of the methods commonly used to evaluate the performance of a classifier.

4.5.1 Holdout Method

In the holdout method, the original data with labeled examples is partitioned into two disjoint sets, called the training and the test sets, respectively. A classification model is then induced from the training set and its performance is evaluated on the test set. The proportion of data reserved for training and for testing is typically at the discretion of the analysts (e.g., 50-50 or two-thirds for training and one-third for testing). The accuracy of the classifier can be estimated based on the accuracy of the induced model on the test set.

The holdout method has several well-known limitations. First, fewer labeled examples are available for training because some of the records are withheld for testing. As a result, the induced model may not be as good as when all the labeled examples are used for training. Second, the model may be highly dependent on the composition of the training and test sets. The smaller the training set size, the larger the variance of the model. On the other hand, if the training set is too large, then the estimated accuracy computed from the smaller test set is less reliable. Such an estimate is said to have a wide confidence interval. Finally, the training and test sets are no longer independent of each other. Because the training and test sets are subsets of the original data, a class that is overrepresented in one subset will be underrepresented in the other, and vice versa.

4.5.2 Random Subsampling

The holdout method can be repeated several times to improve the estimation of a classifier's performance. This approach is known as random subsampling. Let acc_i be the model accuracy during the i^{th} iteration. The overall accuracy is given by $acc_{\text{sub}} = \sum_{i=1}^{k} acc_i/k$. Random subsampling still encounters some of the problems associated with the holdout method because it does not utilize as much data as possible for training. It also has no control over the number of times each record is used for testing and training. Consequently, some records might be used for training more often than others.

4.5.3 Cross-Validation

An alternative to random subsampling is cross-validation. In this approach, each record is used the same number of times for training and exactly once for testing. To illustrate this method, suppose we partition the data into two equal-sized subsets. First, we choose one of the subsets for training and the other for testing. We then swap the roles of the subsets so that the previous training set becomes the test set and vice versa. This approach is called a two-fold cross-validation. The total error is obtained by summing up the errors for both runs. In this example, each record is used exactly once for training and once for testing. The k-fold cross-validation method generalizes this approach by segmenting the data into k equal-sized partitions. During each run, one of the partitions is chosen for testing, while the rest of them are used for training. This procedure is repeated k times so that each partition is used for testing exactly once. Again, the total error is found by summing up the errors for all k runs. A special case of the k-fold cross-validation method sets $k = N$, the size of the data set. In this so-called **leave-one-out** approach, each test set contains only one record. This approach has the advantage of utilizing as much data as possible for training. In addition, the test sets are mutually exclusive and they effectively cover the entire data set. The drawback of this approach is that it is computationally expensive to repeat the procedure N times. Furthermore, since each test set contains only one record, the variance of the estimated performance metric tends to be high.

4.5.4 Bootstrap

The methods presented so far assume that the training records are sampled without replacement. As a result, there are no duplicate records in the training and test sets. In the bootstrap approach, the training records are sampled with replacement; i.e., a record already chosen for training is put back into the original pool of records so that it is equally likely to be redrawn. If the original data has N records, it can be shown that, on average, a bootstrap sample of size N contains about 63.2% of the records in the original data. This approximation follows from the fact that the probability a record is chosen by a bootstrap sample is $1 - (1 - 1/N)^N$. When N is sufficiently large, the probability asymptotically approaches $1 - e^{-1} = 0.632$. Records that are not included in the bootstrap sample become part of the test set. The model induced from the training set is then applied to the test set to obtain an estimate of the accuracy of the bootstrap sample, ϵ_i. The sampling procedure is then repeated b times to generate b bootstrap samples.

There are several variations to the bootstrap sampling approach in terms of how the overall accuracy of the classifier is computed. One of the more widely used approaches is the **.632 bootstrap**, which computes the overall accuracy by combining the accuracies of each bootstrap sample (ϵ_i) with the accuracy computed from a training set that contains all the labeled examples in the original data (acc_s):

$$\text{Accuracy, } acc_{boot} = \frac{1}{b}\sum_{i=1}^{b}(0.632 \times \epsilon_i + 0.368 \times acc_s). \qquad (4.11)$$

4.6 Methods for Comparing Classifiers

It is often useful to compare the performance of different classifiers to determine which classifier works better on a given data set. However, depending on the size of the data, the observed difference in accuracy between two classifiers may not be statistically significant. This section examines some of the statistical tests available to compare the performance of different models and classifiers.

For illustrative purposes, consider a pair of classification models, M_A and M_B. Suppose M_A achieves 85% accuracy when evaluated on a test set containing 30 records, while M_B achieves 75% accuracy on a different test set containing 5000 records. Based on this information, is M_A a better model than M_B?

The preceding example raises two key questions regarding the statistical significance of the performance metrics:

1. Although M_A has a higher accuracy than M_B, it was tested on a smaller test set. How much confidence can we place on the accuracy for M_A?

2. Is it possible to explain the difference in accuracy as a result of variations in the composition of the test sets?

The first question relates to the issue of estimating the confidence interval of a given model accuracy. The second question relates to the issue of testing the statistical significance of the observed deviation. These issues are investigated in the remainder of this section.

4.6.1 Estimating a Confidence Interval for Accuracy

To determine the confidence interval, we need to establish the probability distribution that governs the accuracy measure. This section describes an approach for deriving the confidence interval by modeling the classification task as a binomial experiment. Following is a list of characteristics of a binomial experiment:

1. The experiment consists of N independent trials, where each trial has two possible outcomes: success or failure.

2. The probability of success, p, in each trial is constant.

An example of a binomial experiment is counting the number of heads that turn up when a coin is flipped N times. If X is the number of successes observed in N trials, then the probability that X takes a particular value is given by a binomial distribution with mean Np and variance $Np(1-p)$:

$$P(X = v) = \binom{N}{p} p^v (1-p)^{N-v}.$$

For example, if the coin is fair ($p = 0.5$) and is flipped fifty times, then the probability that the head shows up 20 times is

$$P(X = 20) = \binom{50}{20} 0.5^{20} (1-0.5)^{30} = 0.0419.$$

If the experiment is repeated many times, then the average number of heads expected to show up is $50 \times 0.5 = 25$, while its variance is $50 \times 0.5 \times 0.5 = 12.5$.

The task of predicting the class labels of test records can also be considered as a binomial experiment. Given a test set that contains N records, let X be the number of records correctly predicted by a model and p be the true accuracy of the model. By modeling the prediction task as a binomial experiment, X has a binomial distribution with mean Np and variance $Np(1-p)$. It can be shown that the empirical accuracy, $acc = X/N$, also has a binomial distribution with mean p and variance $p(1-p)/N$ (see Exercise 12). Although the binomial distribution can be used to estimate the confidence interval for acc, it is often approximated by a normal distribution when N is sufficiently large. Based on the normal distribution, the following confidence interval for acc can be derived:

$$P\Big(-Z_{\alpha/2} \le \frac{acc - p}{\sqrt{p(1-p)/N}} \le Z_{1-\alpha/2}\Big) = 1 - \alpha, \tag{4.12}$$

where $Z_{\alpha/2}$ and $Z_{1-\alpha/2}$ are the upper and lower bounds obtained from a standard normal distribution at confidence level $(1-\alpha)$. Since a standard normal distribution is symmetric around $Z = 0$, it follows that $Z_{\alpha/2} = Z_{1-\alpha/2}$. Rearranging this inequality leads to the following confidence interval for p:

$$\frac{2 \times N \times acc + Z^2_{\alpha/2} \pm Z_{\alpha/2}\sqrt{Z^2_{\alpha/2} + 4Nacc - 4Nacc^2}}{2(N + Z^2_{\alpha/2})}. \tag{4.13}$$

The following table shows the values of $Z_{\alpha/2}$ at different confidence levels:

$1-\alpha$	0.99	0.98	0.95	0.9	0.8	0.7	0.5
$Z_{\alpha/2}$	2.58	2.33	1.96	1.65	1.28	1.04	0.67

Example 4.4. Consider a model that has an accuracy of 80% when evaluated on 100 test records. What is the confidence interval for its true accuracy at a 95% confidence level? The confidence level of 95% corresponds to $Z_{\alpha/2} = 1.96$ according to the table given above. Inserting this term into Equation 4.13 yields a confidence interval between 71.1% and 86.7%. The following table shows the confidence interval when the number of records, N, increases:

N	20	50	100	500	1000	5000
Confidence Interval	0.584 – 0.919	0.670 – 0.888	0.711 – 0.867	0.763 – 0.833	0.774 – 0.824	0.789 – 0.811

Note that the confidence interval becomes tighter when N increases. ■

4.6.2 Comparing the Performance of Two Models

Consider a pair of models, M_1 and M_2, that are evaluated on two independent test sets, D_1 and D_2. Let n_1 denote the number of records in D_1 and n_2 denote the number of records in D_2. In addition, suppose the error rate for M_1 on D_1 is e_1 and the error rate for M_2 on D_2 is e_2. Our goal is to test whether the observed difference between e_1 and e_2 is statistically significant.

Assuming that n_1 and n_2 are sufficiently large, the error rates e_1 and e_2 can be approximated using normal distributions. If the observed difference in the error rate is denoted as $d = e_1 - e_2$, then d is also normally distributed with mean d_t, its true difference, and variance, σ_d^2. The variance of d can be computed as follows:

$$\sigma_d^2 \simeq \widehat{\sigma}_d^2 = \frac{e_1(1-e_1)}{n_1} + \frac{e_2(1-e_2)}{n_2}, \tag{4.14}$$

where $e_1(1-e_1)/n_1$ and $e_2(1-e_2)/n_2$ are the variances of the error rates. Finally, at the $(1-\alpha)\%$ confidence level, it can be shown that the confidence interval for the true difference d_t is given by the following equation:

$$d_t = d \pm z_{\alpha/2}\widehat{\sigma}_d. \tag{4.15}$$

Example 4.5. Consider the problem described at the beginning of this section. Model M_A has an error rate of $e_1 = 0.15$ when applied to $N_1 = 30$ test records, while model M_B has an error rate of $e_2 = 0.25$ when applied to $N_2 = 5000$ test records. The observed difference in their error rates is $d = |0.15 - 0.25| = 0.1$. In this example, we are performing a two-sided test to check whether $d_t = 0$ or $d_t \neq 0$. The estimated variance of the observed difference in error rates can be computed as follows:

$$\widehat{\sigma}_d^2 = \frac{0.15(1-0.15)}{30} + \frac{0.25(1-0.25)}{5000} = 0.0043$$

or $\widehat{\sigma}_d = 0.0655$. Inserting this value into Equation 4.15, we obtain the following confidence interval for d_t at 95% confidence level:

$$d_t = 0.1 \pm 1.96 \times 0.0655 = 0.1 \pm 0.128.$$

As the interval spans the value zero, we can conclude that the observed difference is not statistically significant at a 95% confidence level. ■

At what confidence level can we reject the hypothesis that $d_t = 0$? To do this, we need to determine the value of $Z_{\alpha/2}$ such that the confidence interval for d_t does not span the value zero. We can reverse the preceding computation and look for the value $Z_{\alpha/2}$ such that $d > Z_{\alpha/2}\widehat{\sigma}_d$. Replacing the values of d and $\widehat{\sigma}_d$ gives $Z_{\alpha/2} < 1.527$. This value first occurs when $(1-\alpha) \lesssim 0.936$ (for a two-sided test). The result suggests that the null hypothesis can be rejected at confidence level of 93.6% or lower.

4.6.3 Comparing the Performance of Two Classifiers

Suppose we want to compare the performance of two classifiers using the k-fold cross-validation approach. Initially, the data set D is divided into k equal-sized partitions. We then apply each classifier to construct a model from $k-1$ of the partitions and test it on the remaining partition. This step is repeated k times, each time using a different partition as the test set.

Let M_{ij} denote the model induced by classification technique L_i during the j^{th} iteration. Note that each pair of models M_{1j} and M_{2j} are tested on the same partition j. Let e_{1j} and e_{2j} be their respective error rates. The difference between their error rates during the j^{th} fold can be written as $d_j = e_{1j} - e_{2j}$. If k is sufficiently large, then d_j is normally distributed with mean d_t^{cv}, which is the true difference in their error rates, and variance σ^{cv}. Unlike the previous approach, the overall variance in the observed differences is estimated using the following formula:

$$\widehat{\sigma}^2_{d^{cv}} = \frac{\sum_{j=1}^{k}(d_j - \overline{d})^2}{k(k-1)}, \tag{4.16}$$

where $\overline{d}$ is the average difference. For this approach, we need to use a t-distribution to compute the confidence interval for d_t^{cv}:

$$d_t^{cv} = \overline{d} \pm t_{(1-\alpha),k-1}\widehat{\sigma}_{d^{cv}}.$$

The coefficient $t_{(1-\alpha),k-1}$ is obtained from a probability table with two input parameters, its confidence level $(1-\alpha)$ and the number of degrees of freedom, $k-1$. The probability table for the t-distribution is shown in Table 4.6.

Example 4.6. Suppose the estimated difference in the accuracy of models generated by two classification techniques has a mean equal to 0.05 and a standard deviation equal to 0.002. If the accuracy is estimated using a 30-fold cross-validation approach, then at a 95% confidence level, the true accuracy difference is

$$d_t^{cv} = 0.05 \pm 2.04 \times 0.002. \tag{4.17}$$

Table 4.6. Probability table for t-distribution.

	$(1-\alpha)$				
$k-1$	0.99	0.98	0.95	0.9	0.8
1	3.08	6.31	12.7	31.8	63.7
2	1.89	2.92	4.30	6.96	9.92
4	1.53	2.13	2.78	3.75	4.60
9	1.38	1.83	2.26	2.82	3.25
14	1.34	1.76	2.14	2.62	2.98
19	1.33	1.73	2.09	2.54	2.86
24	1.32	1.71	2.06	2.49	2.80
29	1.31	1.70	2.04	2.46	2.76

Since the confidence interval does not span the value zero, the observed difference between the techniques is statistically significant. ■

4.7 Bibliographic Notes

Early classification systems were developed to organize a large collection of objects. For example, the Dewey Decimal and Library of Congress classification systems were designed to catalog and index the vast number of library books. The categories are typically identified in a manual fashion, with the help of domain experts.

Automated classification has been a subject of intensive research for many years. The study of classification in classical statistics is sometimes known as **discriminant analysis**, where the objective is to predict the group membership of an object based on a set of predictor variables. A well-known classical method is Fisher's linear discriminant analysis [117], which seeks to find a linear projection of the data that produces the greatest discrimination between objects that belong to different classes.

Many pattern recognition problems also require the discrimination of objects from different classes. Examples include speech recognition, handwritten character identification, and image classification. Readers who are interested in the application of classification techniques for pattern recognition can refer to the survey articles by Jain et al. [122] and Kulkarni et al. [128] or classic pattern recognition books by Bishop [107], Duda et al. [114], and Fukunaga [118]. The subject of classification is also a major research topic in the fields of neural networks, statistical learning, and machine learning. An in-depth treatment of various classification techniques is given in the books by Cherkassky and Mulier [112], Hastie et al. [120], Michie et al. [133], and Mitchell [136].

An overview of decision tree induction algorithms can be found in the survey articles by Buntine [110], Moret [137], Murthy [138], and Safavian et al. [147]. Examples of some well-known decision tree algorithms include CART [108], ID3 [143], C4.5 [145], and CHAID [125]. Both ID3 and C4.5 employ the entropy measure as their splitting function. An in-depth discussion of the C4.5 decision tree algorithm is given by Quinlan [145]. Besides explaining the methodology for decision tree growing and tree pruning, Quinlan [145] also described how the algorithm can be modified to handle data sets with missing values. The CART algorithm was developed by Breiman et al. [108] and uses the Gini index as its splitting function. CHAID [125] uses the statistical χ^2 test to determine the best split during the tree-growing process.

The decision tree algorithm presented in this chapter assumes that the splitting condition is specified one attribute at a time. An oblique decision tree can use multiple attributes to form the attribute test condition in the internal nodes [121, 152]. Breiman et al. [108] provide an option for using linear combinations of attributes in their CART implementation. Other approaches for inducing oblique decision trees were proposed by Heath et al. [121], Murthy et al. [139], Cantú-Paz and Kamath [111], and Utgoff and Brodley [152]. Although oblique decision trees help to improve the expressiveness of a decision tree representation, learning the appropriate test condition at each node is computationally challenging. Another way to improve the expressiveness of a decision tree without using oblique decision trees is to apply a method known as **constructive induction** [132]. This method simplifies the task of learning complex splitting functions by creating compound features from the original attributes.

Besides the top-down approach, other strategies for growing a decision tree include the bottom-up approach by Landeweerd et al. [130] and Pattipati and Alexandridis [142], as well as the bidirectional approach by Kim and Landgrebe [126]. Schuermann and Doster [150] and Wang and Suen [154] proposed using a **soft splitting criterion** to address the data fragmentation problem. In this approach, each record is assigned to different branches of the decision tree with different probabilities.

Model overfitting is an important issue that must be addressed to ensure that a decision tree classifier performs equally well on previously unknown records. The model overfitting problem has been investigated by many authors including Breiman et al. [108], Schaffer [148], Mingers [135], and Jensen and Cohen [123]. While the presence of noise is often regarded as one of the primary reasons for overfitting [135, 140], Jensen and Cohen [123] argued that overfitting is the result of using incorrect hypothesis tests in a multiple comparison procedure.

Schapire [149] defined generalization error as “the probability of misclassifying a new example” and test error as “the fraction of mistakes on a newly sampled test set.” Generalization error can therefore be considered as the expected test error of a classifier. Generalization error may sometimes refer to the true error [136] of a model, i.e., its expected error for randomly drawn data points from the same population distribution where the training set is sampled. These definitions are in fact equivalent if both the training and test sets are gathered from the same population distribution, which is often the case in many data mining and machine learning applications.

The Occam’s razor principle is often attributed to the philosopher William of Occam. Domingos [113] cautioned against the pitfall of misinterpreting Occam’s razor as comparing models with similar training errors, instead of generalization errors. A survey on decision tree-pruning methods to avoid overfitting is given by Breslow and Aha [109] and Esposito et al. [116]. Some of the typical pruning methods include reduced error pruning [144], pessimistic error pruning [144], minimum error pruning [141], critical value pruning [134], cost-complexity pruning [108], and error-based pruning [145]. Quinlan and Rivest proposed using the minimum description length principle for decision tree pruning in [146].

Kohavi [127] had performed an extensive empirical study to compare the performance metrics obtained using different estimation methods such as random subsampling, bootstrapping, and k-fold cross-validation. Their results suggest that the best estimation method is based on the ten-fold stratified cross-validation. Efron and Tibshirani [115] provided a theoretical and empirical comparison between cross-validation and a bootstrap method known as the 632+ rule.

Current techniques such as C4.5 require that the entire training data set fit into main memory. There has been considerable effort to develop parallel and scalable versions of decision tree induction algorithms. Some of the proposed algorithms include SLIQ by Mehta et al. [131], SPRINT by Shafer et al. [151], CMP by Wang and Zaniolo [153], CLOUDS by Alsabti et al. [106], RainForest by Gehrke et al. [119], and ScalParC by Joshi et al. [124]. A general survey of parallel algorithms for data mining is available in [129].

Bibliography

[106] K. Alsabti, S. Ranka, and V. Singh. CLOUDS: A Decision Tree Classifier for Large Datasets. In *Proc. of the 4th Intl. Conf. on Knowledge Discovery and Data Mining*, pages 2–8, New York, NY, August 1998.

[107] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Oxford, U.K., 1995.

[108] L. Breiman, J. H. Friedman, R. Olshen, and C. J. Stone. *Classification and Regression Trees.* Chapman & Hall, New York, 1984.

[109] L. A. Breslow and D. W. Aha. Simplifying Decision Trees: A Survey. *Knowledge Engineering Review*, 12(1):1–40, 1997.

[110] W. Buntine. Learning classification trees. In *Artificial Intelligence Frontiers in Statistics*, pages 182–201. Chapman & Hall, London, 1993.

[111] E. Cantú-Paz and C. Kamath. Using evolutionary algorithms to induce oblique decision trees. In *Proc. of the Genetic and Evolutionary Computation Conf.*, pages 1053–1060, San Francisco, CA, 2000.

[112] V. Cherkassky and F. Mulier. *Learning from Data: Concepts, Theory, and Methods.* Wiley Interscience, 1998.

[113] P. Domingos. The Role of Occam's Razor in Knowledge Discovery. *Data Mining and Knowledge Discovery*, 3(4):409–425, 1999.

[114] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification.* John Wiley & Sons, Inc., New York, 2nd edition, 2001.

[115] B. Efron and R. Tibshirani. Cross-validation and the Bootstrap: Estimating the Error Rate of a Prediction Rule. Technical report, Stanford University, 1995.

[116] F. Esposito, D. Malerba, and G. Semeraro. A Comparative Analysis of Methods for Pruning Decision Trees. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 19 (5):476–491, May 1997.

[117] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.

[118] K. Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, New York, 1990.

[119] J. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest—A Framework for Fast Decision Tree Construction of Large Datasets. *Data Mining and Knowledge Discovery*, 4 (2/3):127–162, 2000.

[120] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, Prediction.* Springer, New York, 2001.

[121] D. Heath, S. Kasif, and S. Salzberg. Induction of Oblique Decision Trees. In *Proc. of the 13th Intl. Joint Conf. on Artificial Intelligence*, pages 1002–1007, Chambery, France, August 1993.

[122] A. K. Jain, R. P. W. Duin, and J. Mao. Statistical Pattern Recognition: A Review. *IEEE Tran. Patt. Anal. and Mach. Intellig.*, 22(1):4–37, 2000.

[123] D. Jensen and P. R. Cohen. Multiple Comparisons in Induction Algorithms. *Machine Learning*, 38(3):309–338, March 2000.

[124] M. V. Joshi, G. Karypis, and V. Kumar. ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets. In *Proc. of 12th Intl. Parallel Processing Symp. (IPPS/SPDP)*, pages 573–579, Orlando, FL, April 1998.

[125] G. V. Kass. An Exploratory Technique for Investigating Large Quantities of Categorical Data. *Applied Statistics*, 29:119–127, 1980.

[126] B. Kim and D. Landgrebe. Hierarchical decision classifiers in high-dimensional and large class data. *IEEE Trans. on Geoscience and Remote Sensing*, 29(4):518–528, 1991.

[127] R. Kohavi. A Study on Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proc. of the 15th Intl. Joint Conf. on Artificial Intelligence*, pages 1137–1145, Montreal, Canada, August 1995.

[128] S. R. Kulkarni, G. Lugosi, and S. S. Venkatesh. Learning Pattern Classification—A Survey. *IEEE Tran. Inf. Theory*, 44(6):2178–2206, 1998.

[129] V. Kumar, M. V. Joshi, E.-H. Han, P. N. Tan, and M. Steinbach. High Performance Data Mining. In *High Performance Computing for Computational Science (VECPAR 2002)*, pages 111–125. Springer, 2002.

[130] G. Landeweerd, T. Timmers, E. Gersema, M. Bins, and M. Halic. Binary tree versus single level tree classification of white blood cells. *Pattern Recognition*, 16:571–577, 1983.

[131] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A Fast Scalable Classifier for Data Mining. In *Proc. of the 5th Intl. Conf. on Extending Database Technology*, pages 18–32, Avignon, France, March 1996.

[132] R. S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–116, 1983.

[133] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, Upper Saddle River, NJ, 1994.

[134] J. Mingers. Expert Systems—Rule Induction with Statistical Data. *J Operational Research Society*, 38:39–47, 1987.

[135] J. Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine Learning*, 4:227–243, 1989.

[136] T. Mitchell. *Machine Learning*. McGraw-Hill, Boston, MA, 1997.

[137] B. M. E. Moret. Decision Trees and Diagrams. *Computing Surveys*, 14(4):593–623, 1982.

[138] S. K. Murthy. Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, 1998.

[139] S. K. Murthy, S. Kasif, and S. Salzberg. A system for induction of oblique decision trees. *J of Artificial Intelligence Research*, 2:1–33, 1994.

[140] T. Niblett. Constructing decision trees in noisy domains. In *Proc. of the 2nd European Working Session on Learning*, pages 67–78, Bled, Yugoslavia, May 1987.

[141] T. Niblett and I. Bratko. Learning Decision Rules in Noisy Domains. In *Research and Development in Expert Systems III*, Cambridge, 1986. Cambridge University Press.

[142] K. R. Pattipati and M. G. Alexandridis. Application of heuristic search and information theory to sequential fault diagnosis. *IEEE Trans. on Systems, Man, and Cybernetics*, 20(4):872–887, 1990.

[143] J. R. Quinlan. Discovering rules by induction from large collection of examples. In D. Michie, editor, *Expert Systems in the Micro Electronic Age*. Edinburgh University Press, Edinburgh, UK, 1979.

[144] J. R. Quinlan. Simplifying Decision Trees. *Intl. J. Man-Machine Studies*, 27:221–234, 1987.

[145] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan-Kaufmann Publishers, San Mateo, CA, 1993.

[146] J. R. Quinlan and R. L. Rivest. Inferring Decision Trees Using the Minimum Description Length Principle. *Information and Computation*, 80(3):227–248, 1989.

[147] S. R. Safavian and D. Landgrebe. A Survey of Decision Tree Classifier Methodology. *IEEE Trans. Systems, Man and Cybernetics*, 22:660–674, May/June 1998.

[148] C. Schaffer. Overfitting avoidence as bias. *Machine Learning*, 10:153–178, 1993.

[149] R. E. Schapire. The Boosting Approach to Machine Learning: An Overview. In *MSRI Workshop on Nonlinear Estimation and Classification*, 2002.

[150] J. Schuermann and W. Doster. A decision-theoretic approach in hierarchical classifier design. *Pattern Recognition*, 17:359–369, 1984.

[151] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A Scalable Parallel Classifier for Data Mining. In *Proc. of the 22nd VLDB Conf.*, pages 544–555, Bombay, India, September 1996.

[152] P. E. Utgoff and C. E. Brodley. An incremental method for finding multivariate splits for decision trees. In *Proc. of the 7th Intl. Conf. on Machine Learning*, pages 58–65, Austin, TX, June 1990.

[153] H. Wang and C. Zaniolo. CMP: A Fast Decision Tree Classifier Using Multivariate Predictions. In *Proc. of the 16th Intl. Conf. on Data Engineering*, pages 449–460, San Diego, CA, March 2000.

[154] Q. R. Wang and C. Y. Suen. Large tree classifier with heuristic search and global training. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 9(1):91–102, 1987.

4.8 Exercises

1. Draw the full decision tree for the parity function of four Boolean attributes, A, B, C, and D. Is it possible to simplify the tree?

2. Consider the training examples shown in Table 4.7 for a binary classification problem.

   (a) Compute the Gini index for the overall collection of training examples.

   (b) Compute the Gini index for the `Customer ID` attribute.

   (c) Compute the Gini index for the `Gender` attribute.

   (d) Compute the Gini index for the `Car Type` attribute using multiway split.

   (e) Compute the Gini index for the `Shirt Size` attribute using multiway split.

   (f) Which attribute is better, `Gender`, `Car Type`, or `Shirt Size`?

   (g) Explain why `Customer ID` should not be used as the attribute test condition even though it has the lowest Gini.

3. Consider the training examples shown in Table 4.8 for a binary classification problem.

   (a) What is the entropy of this collection of training examples with respect to the positive class?

Table 4.7. Data set for Exercise 2.

Customer ID	Gender	Car Type	Shirt Size	Class
1	M	Family	Small	C0
2	M	Sports	Medium	C0
3	M	Sports	Medium	C0
4	M	Sports	Large	C0
5	M	Sports	Extra Large	C0
6	M	Sports	Extra Large	C0
7	F	Sports	Small	C0
8	F	Sports	Small	C0
9	F	Sports	Medium	C0
10	F	Luxury	Large	C0
11	M	Family	Large	C1
12	M	Family	Extra Large	C1
13	M	Family	Medium	C1
14	M	Luxury	Extra Large	C1
15	F	Luxury	Small	C1
16	F	Luxury	Small	C1
17	F	Luxury	Medium	C1
18	F	Luxury	Medium	C1
19	F	Luxury	Medium	C1
20	F	Luxury	Large	C1

Table 4.8. Data set for Exercise 3.

Instance	a_1	a_2	a_3	Target Class
1	T	T	1.0	+
2	T	T	6.0	+
3	T	F	5.0	−
4	F	F	4.0	+
5	F	T	7.0	−
6	F	T	3.0	−
7	F	F	8.0	−
8	T	F	7.0	+
9	F	T	5.0	−

(b) What are the information gains of a_1 and a_2 relative to these training examples?

(c) For a_3, which is a continuous attribute, compute the information gain for every possible split.

(d) What is the best split (among a_1, a_2, and a_3) according to the information gain?

(e) What is the best split (between a_1 and a_2) according to the classification error rate?

(f) What is the best split (between a_1 and a_2) according to the Gini index?

4. Show that the entropy of a node never increases after splitting it into smaller successor nodes.

5. Consider the following data set for a binary class problem.

A	B	Class Label
T	F	+
T	T	+
T	T	+
T	F	−
T	T	+
F	F	−
F	F	−
F	F	−
T	T	−
T	F	−

(a) Calculate the information gain when splitting on A and B. Which attribute would the decision tree induction algorithm choose?

(b) Calculate the gain in the Gini index when splitting on A and B. Which attribute would the decision tree induction algorithm choose?

(c) Figure 4.13 shows that entropy and the Gini index are both monotonously increasing on the range [0, 0.5] and they are both monotonously decreasing on the range [0.5, 1]. Is it possible that information gain and the gain in the Gini index favor different attributes? Explain.

6. Consider the following set of training examples.

X	Y	Z	No. of Class C1 Examples	No. of Class C2 Examples
0	0	0	5	40
0	0	1	0	15
0	1	0	10	5
0	1	1	45	0
1	0	0	10	5
1	0	1	25	0
1	1	0	5	20
1	1	1	0	15

(a) Compute a two-level decision tree using the greedy approach described in this chapter. Use the classification error rate as the criterion for splitting. What is the overall error rate of the induced tree?

(b) Repeat part (a) using X as the first splitting attribute and then choose the best remaining attribute for splitting at each of the two successor nodes. What is the error rate of the induced tree?

(c) Compare the results of parts (a) and (b). Comment on the suitability of the greedy heuristic used for splitting attribute selection.

7. The following table summarizes a data set with three attributes A, B, C and two class labels $+$, $-$. Build a two-level decision tree.

A	B	C	Number of Instances	
			$+$	$-$
T	T	T	5	0
F	T	T	0	20
T	F	T	20	0
F	F	T	0	5
T	T	F	0	0
F	T	F	25	0
T	F	F	0	0
F	F	F	0	25

(a) According to the classification error rate, which attribute would be chosen as the first splitting attribute? For each attribute, show the contingency table and the gains in classification error rate.

(b) Repeat for the two children of the root node.

(c) How many instances are misclassified by the resulting decision tree?

(d) Repeat parts (a), (b), and (c) using C as the splitting attribute.

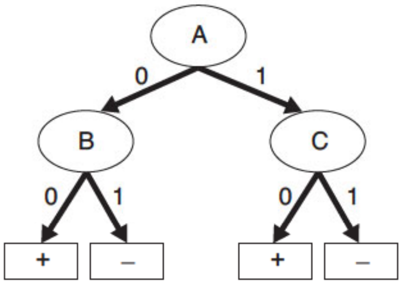
(e) Use the results in parts (c) and (d) to conclude about the greedy nature of the decision tree induction algorithm.

8. Consider the decision tree shown in Figure 4.30.

(a) Compute the generalization error rate of the tree using the optimistic approach.

(b) Compute the generalization error rate of the tree using the pessimistic approach. (For simplicity, use the strategy of adding a factor of 0.5 to each leaf node.)

(c) Compute the generalization error rate of the tree using the validation set shown above. This approach is known as **reduced error pruning**.



Training:

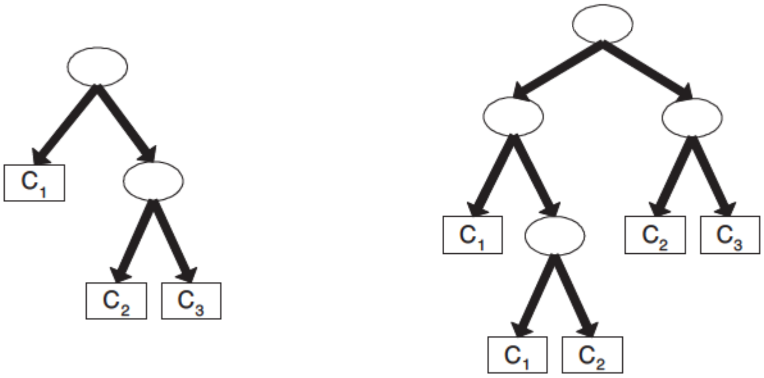
Instance	A	B	C	Class
1	0	0	0	+
2	0	0	1	+
3	0	1	0	+
4	0	1	1	–
5	1	0	0	+
6	1	0	0	+
7	1	1	0	–
8	1	0	1	+
9	1	1	0	–
10	1	1	0	–

Validation:

Instance	A	B	C	Class
11	0	0	0	+
12	0	1	1	+
13	1	1	0	+
14	1	0	1	–
15	1	0	0	+

Figure 4.30. Decision tree and data sets for Exercise 8.

9. Consider the decision trees shown in Figure 4.31. Assume they are generated from a data set that contains 16 binary attributes and 3 classes, C_1, C_2, and C_3.



(a) Decision tree with 7 errors

(b) Decision tree with 4 errors

Figure 4.31. Decision trees for Exercise 9.

Compute the total description length of each decision tree according to the minimum description length principle.

- The total description length of a tree is given by:

$$Cost(tree, data) = Cost(tree) + Cost(data|tree).$$

- Each internal node of the tree is encoded by the ID of the splitting attribute. If there are m attributes, the cost of encoding each attribute is $\log_2 m$ bits.
- Each leaf is encoded using the ID of the class it is associated with. If there are k classes, the cost of encoding a class is $\log_2 k$ bits.
- $Cost(tree)$ is the cost of encoding all the nodes in the tree. To simplify the computation, you can assume that the total cost of the tree is obtained by adding up the costs of encoding each internal node and each leaf node.
- $Cost(data|tree)$ is encoded using the classification errors the tree commits on the training set. Each error is encoded by $\log_2 n$ bits, where n is the total number of training instances.

Which decision tree is better, according to the MDL principle?

10. While the .632 bootstrap approach is useful for obtaining a reliable estimate of model accuracy, it has a known limitation [127]. Consider a two-class problem, where there are equal number of positive and negative examples in the data. Suppose the class labels for the examples are generated randomly. The classifier used is an unpruned decision tree (i.e., a perfect memorizer). Determine the accuracy of the classifier using each of the following methods.

    (a) The holdout method, where two-thirds of the data are used for training and the remaining one-third are used for testing.

    (b) Ten-fold cross-validation.

    (c) The .632 bootstrap method.

    (d) From the results in parts (a), (b), and (c), which method provides a more reliable evaluation of the classifier's accuracy?

11. Consider the following approach for testing whether a classifier A beats another classifier B. Let N be the size of a given data set, p_A be the accuracy of classifier A, p_B be the accuracy of classifier B, and $p = (p_A + p_B)/2$ be the average accuracy for both classifiers. To test whether classifier A is significantly better than B, the following Z-statistic is used:

$$Z = \frac{p_A - p_B}{\sqrt{\frac{2p(1-p)}{N}}}.$$

Classifier A is assumed to be better than classifier B if Z > 1.96.

Table 4.9 compares the accuracies of three different classifiers, decision tree classifiers, naïve Bayes classifiers, and support vector machines, on various data sets. (The latter two classifiers are described in Chapter 5.)

Table 4.9. Comparing the accuracy of various classification methods.

Data Set	Size (N)	Decision Tree (%)	naïve Bayes (%)	Support vector machine (%)
Anneal	898	92.09	79.62	87.19
Australia	690	85.51	76.81	84.78
Auto	205	81.95	58.05	70.73
Breast	699	95.14	95.99	96.42
Cleve	303	76.24	83.50	84.49
Credit	690	85.80	77.54	85.07
Diabetes	768	72.40	75.91	76.82
German	1000	70.90	74.70	74.40
Glass	214	67.29	48.59	59.81
Heart	270	80.00	84.07	83.70
Hepatitis	155	81.94	83.23	87.10
Horse	368	85.33	78.80	82.61
Ionosphere	351	89.17	82.34	88.89
Iris	150	94.67	95.33	96.00
Labor	57	78.95	94.74	92.98
Led7	3200	73.34	73.16	73.56
Lymphography	148	77.03	83.11	86.49
Pima	768	74.35	76.04	76.95
Sonar	208	78.85	69.71	76.92
Tic-tac-toe	958	83.72	70.04	98.33
Vehicle	846	71.04	45.04	74.94
Wine	178	94.38	96.63	98.88
Zoo	101	93.07	93.07	96.04

Summarize the performance of the classifiers given in Table 4.9 using the following 3 × 3 table:

win-loss-draw	Decision tree	Naïve Bayes	Support vector machine
Decision tree	0 - 0 - 23		
Naïve Bayes		0 - 0 - 23	
Support vector machine			0 - 0 - 23

Each cell in the table contains the number of wins, losses, and draws when comparing the classifier in a given row to the classifier in a given column.

12. Let X be a binomial random variable with mean Np and variance $Np(1-p)$. Show that the ratio X/N also has a binomial distribution with mean p and variance $p(1-p)/N$.

6

Association Analysis: Basic Concepts and Algorithms

Many business enterprises accumulate large quantities of data from their day-to-day operations. For example, huge amounts of customer purchase data are collected daily at the checkout counters of grocery stores. Table 6.1 illustrates an example of such data, commonly known as **market basket transactions**. Each row in this table corresponds to a transaction, which contains a unique identifier labeled TID and a set of items bought by a given customer. Retailers are interested in analyzing the data to learn about the purchasing behavior of their customers. Such valuable information can be used to support a variety of business-related applications such as marketing promotions, inventory management, and customer relationship management.

This chapter presents a methodology known as **association analysis**, which is useful for discovering interesting relationships hidden in large data sets. The uncovered relationships can be represented in the form of **associa-**

Table 6.1. An example of market basket transactions.

TID	Items
1	{Bread, Milk}
2	{Bread, Diapers, Beer, Eggs}
3	{Milk, Diapers, Beer, Cola}
4	{Bread, Milk, Diapers, Beer}
5	{Bread, Milk, Diapers, Cola}

tion rules or sets of frequent items. For example, the following rule can be extracted from the data set shown in Table 6.1:

$$\{\texttt{Diapers}\} \longrightarrow \{\texttt{Beer}\}.$$

The rule suggests that a strong relationship exists between the sale of diapers and beer because many customers who buy diapers also buy beer. Retailers can use this type of rules to help them identify new opportunities for cross-selling their products to the customers.

Besides market basket data, association analysis is also applicable to other application domains such as bioinformatics, medical diagnosis, Web mining, and scientific data analysis. In the analysis of Earth science data, for example, the association patterns may reveal interesting connections among the ocean, land, and atmospheric processes. Such information may help Earth scientists develop a better understanding of how the different elements of the Earth system interact with each other. Even though the techniques presented here are generally applicable to a wider variety of data sets, for illustrative purposes, our discussion will focus mainly on market basket data.

There are two key issues that need to be addressed when applying association analysis to market basket data. First, discovering patterns from a large transaction data set can be computationally expensive. Second, some of the discovered patterns are potentially spurious because they may happen simply by chance. The remainder of this chapter is organized around these two issues. The first part of the chapter is devoted to explaining the basic concepts of association analysis and the algorithms used to efficiently mine such patterns. The second part of the chapter deals with the issue of evaluating the discovered patterns in order to prevent the generation of spurious results.

6.1 Problem Definition

This section reviews the basic terminology used in association analysis and presents a formal description of the task.

Binary Representation Market basket data can be represented in a binary format as shown in Table 6.2, where each row corresponds to a transaction and each column corresponds to an item. An item can be treated as a binary variable whose value is one if the item is present in a transaction and zero otherwise. Because the presence of an item in a transaction is often considered more important than its absence, an item is an **asymmetric** binary variable.

Table 6.2. A binary $0/1$ representation of market basket data.

TID	Bread	Milk	Diapers	Beer	Eggs	Cola
1	1	1	0	0	0	0
2	1	0	1	1	1	0
3	0	1	1	1	0	1
4	1	1	1	1	0	0
5	1	1	1	0	0	1

This representation is perhaps a very simplistic view of real market basket data because it ignores certain important aspects of the data such as the quantity of items sold or the price paid to purchase them. Methods for handling such non-binary data will be explained in Chapter 7.

Itemset and Support Count Let $I = \{i_1,i_2,\ldots,i_d\}$ be the set of all items in a market basket data and $T = \{t_1, t_2, \ldots, t_N\}$ be the set of all transactions. Each transaction t_i contains a subset of items chosen from I. In association analysis, a collection of zero or more items is termed an itemset. If an itemset contains k items, it is called a k-itemset. For instance, {`Beer`, `Diapers`, `Milk`} is an example of a 3-itemset. The null (or empty) set is an itemset that does not contain any items.

The transaction width is defined as the number of items present in a transaction. A transaction t_j is said to contain an itemset X if X is a subset of t_j. For example, the second transaction shown in Table 6.2 contains the itemset {`Bread`, `Diapers`} but not {`Bread`, `Milk`}. An important property of an itemset is its support count, which refers to the number of transactions that contain a particular itemset. Mathematically, the support count, $\sigma(X)$, for an itemset X can be stated as follows:

$$\sigma(X) = \big|\{t_i | X \subseteq t_i,\ t_i \in T\}\big|,$$

where the symbol $|\cdot|$ denote the number of elements in a set. In the data set shown in Table 6.2, the support count for {`Beer`, `Diapers`, `Milk`} is equal to two because there are only two transactions that contain all three items.

Association Rule An association rule is an implication expression of the form $X \longrightarrow Y$, where X and Y are disjoint itemsets, i.e., $X \cap Y = \emptyset$. The strength of an association rule can be measured in terms of its **support** and **confidence**. Support determines how often a rule is applicable to a given data set, while confidence determines how frequently items in Y appear in transactions that contain X. The formal definitions of these metrics are

$$\begin{aligned}\text{Support, } s(X \longrightarrow Y) &= \frac{\sigma(X \cup Y)}{N}; & (6.1)\\ \text{Confidence, } c(X \longrightarrow Y) &= \frac{\sigma(X \cup Y)}{\sigma(X)}. & (6.2)\end{aligned}$$

Example 6.1. Consider the rule {`Milk`, `Diapers`} ⟶ {`Beer`}. Since the support count for {`Milk`, `Diapers`, `Beer`} is 2 and the total number of transactions is 5, the rule's support is $2/5 = 0.4$. The rule's confidence is obtained by dividing the support count for {`Milk`, `Diapers`, `Beer`} by the support count for {`Milk`, `Diapers`}. Since there are 3 transactions that contain milk and diapers, the confidence for this rule is $2/3 = 0.67$. ■

Why Use Support and Confidence? Support is an important measure because a rule that has very low support may occur simply by chance. A low support rule is also likely to be uninteresting from a business perspective because it may not be profitable to promote items that customers seldom buy together (with the exception of the situation described in Section 6.8). For these reasons, support is often used to eliminate uninteresting rules. As will be shown in Section 6.2.1, support also has a desirable property that can be exploited for the efficient discovery of association rules.

Confidence, on the other hand, measures the reliability of the inference made by a rule. For a given rule $X \longrightarrow Y$, the higher the confidence, the more likely it is for Y to be present in transactions that contain X. Confidence also provides an estimate of the conditional probability of Y given X.

Association analysis results should be interpreted with caution. The inference made by an association rule does not necessarily imply causality. Instead, it suggests a strong co-occurrence relationship between items in the antecedent and consequent of the rule. Causality, on the other hand, requires knowledge about the causal and effect attributes in the data and typically involves relationships occurring over time (e.g., ozone depletion leads to global warming).

Formulation of Association Rule Mining Problem The association rule mining problem can be formally stated as follows:

Definition 6.1 (Association Rule Discovery). Given a set of transactions T, find all the rules having support $\geq minsup$ and confidence $\geq minconf$, where $minsup$ and $minconf$ are the corresponding support and confidence thresholds.

A brute-force approach for mining association rules is to compute the support and confidence for every possible rule. This approach is prohibitively expensive because there are exponentially many rules that can be extracted from a data set. More specifically, the total number of possible rules extracted from a data set that contains d items is

$$R = 3^d - 2^{d+1} + 1. \tag{6.3}$$

The proof for this equation is left as an exercise to the readers (see Exercise 5 on page 405). Even for the small data set shown in Table 6.1, this approach requires us to compute the support and confidence for $3^6 - 2^7 + 1 = 602$ rules. More than 80% of the rules are discarded after applying $minsup = 20\%$ and $minconf = 50\%$, thus making most of the computations become wasted. To avoid performing needless computations, it would be useful to prune the rules early without having to compute their support and confidence values.

An initial step toward improving the performance of association rule mining algorithms is to decouple the support and confidence requirements. From Equation 6.2, notice that the support of a rule $X \longrightarrow Y$ depends only on the support of its corresponding itemset, $X \cup Y$. For example, the following rules have identical support because they involve items from the same itemset, {Beer, Diapers, Milk}:

$$\begin{array}{ll}\{\texttt{Beer, Diapers}\} \longrightarrow \{\texttt{Milk}\}, & \{\texttt{Beer, Milk}\} \longrightarrow \{\texttt{Diapers}\}, \\ \{\texttt{Diapers, Milk}\} \longrightarrow \{\texttt{Beer}\}, & \{\texttt{Beer}\} \longrightarrow \{\texttt{Diapers, Milk}\}, \\ \{\texttt{Milk}\} \longrightarrow \{\texttt{Beer,Diapers}\}, & \{\texttt{Diapers}\} \longrightarrow \{\texttt{Beer,Milk}\}.\end{array}$$

If the itemset is infrequent, then all six candidate rules can be pruned immediately without our having to compute their confidence values.

Therefore, a common strategy adopted by many association rule mining algorithms is to decompose the problem into two major subtasks:

1. **Frequent Itemset Generation**, whose objective is to find all the itemsets that satisfy the *minsup* threshold. These itemsets are called frequent itemsets.

2. **Rule Generation**, whose objective is to extract all the high-confidence rules from the frequent itemsets found in the previous step. These rules are called strong rules.

The computational requirements for frequent itemset generation are generally more expensive than those of rule generation. Efficient techniques for generating frequent itemsets and association rules are discussed in Sections 6.2 and 6.3, respectively.

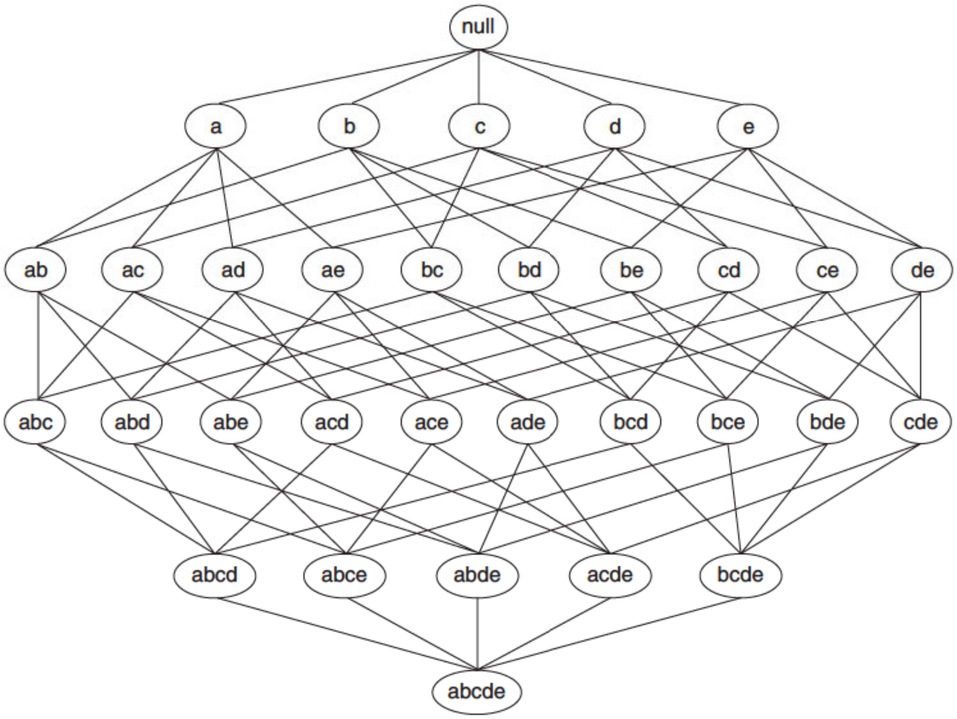
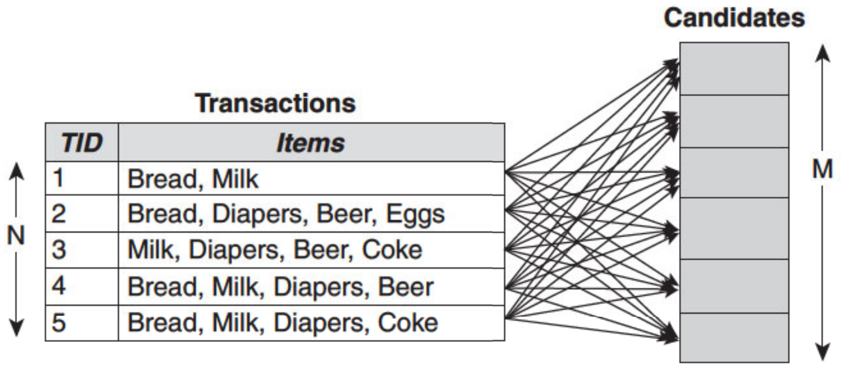


Figure 6.1. An itemset lattice.

6.2 Frequent Itemset Generation

A lattice structure can be used to enumerate the list of all possible itemsets. Figure 6.1 shows an itemset lattice for $I = \{a, b, c, d, e\}$. In general, a data set that contains k items can potentially generate up to $2^k - 1$ frequent itemsets, excluding the null set. Because k can be very large in many practical applications, the search space of itemsets that need to be explored is exponentially large.

A brute-force approach for finding frequent itemsets is to determine the support count for every **candidate itemset** in the lattice structure. To do this, we need to compare each candidate against every transaction, an operation that is shown in Figure 6.2. If the candidate is contained in a transaction, its support count will be incremented. For example, the support for {`Bread`, `Milk`} is incremented three times because the itemset is contained in transactions 1, 4, and 5. Such an approach can be very expensive because it requires $O(NMw)$ comparisons, where N is the number of transactions, $M = 2^k - 1$ is the number of candidate itemsets, and w is the maximum transaction width.



TID	Items
1	Bread, Milk
2	Bread, Diapers, Beer, Eggs
3	Milk, Diapers, Beer, Coke
4	Bread, Milk, Diapers, Beer
5	Bread, Milk, Diapers, Coke

Figure 6.2. Counting the support of candidate itemsets.

There are several ways to reduce the computational complexity of frequent itemset generation.

1. **Reduce the number of candidate itemsets (M).** The *Apriori* principle, described in the next section, is an effective way to eliminate some of the candidate itemsets without counting their support values.

2. **Reduce the number of comparisons.** Instead of matching each candidate itemset against every transaction, we can reduce the number of comparisons by using more advanced data structures, either to store the candidate itemsets or to compress the data set. We will discuss these strategies in Sections 6.2.4 and 6.6.

6.2.1 The *Apriori* Principle

This section describes how the support measure helps to reduce the number of candidate itemsets explored during frequent itemset generation. The use of support for pruning candidate itemsets is guided by the following principle.

Theorem 6.1 (*Apriori* Principle). *If an itemset is frequent, then all of its subsets must also be frequent.*

To illustrate the idea behind the *Apriori* principle, consider the itemset lattice shown in Figure 6.3. Suppose $\{c, d, e\}$ is a frequent itemset. Clearly, any transaction that contains $\{c, d, e\}$ must also contain its subsets, $\{c, d\}$, $\{c, e\}$, $\{d, e\}$, $\{c\}$, $\{d\}$, and $\{e\}$. As a result, if $\{c, d, e\}$ is frequent, then all subsets of $\{c, d, e\}$ (i.e., the shaded itemsets in this figure) must also be frequent.

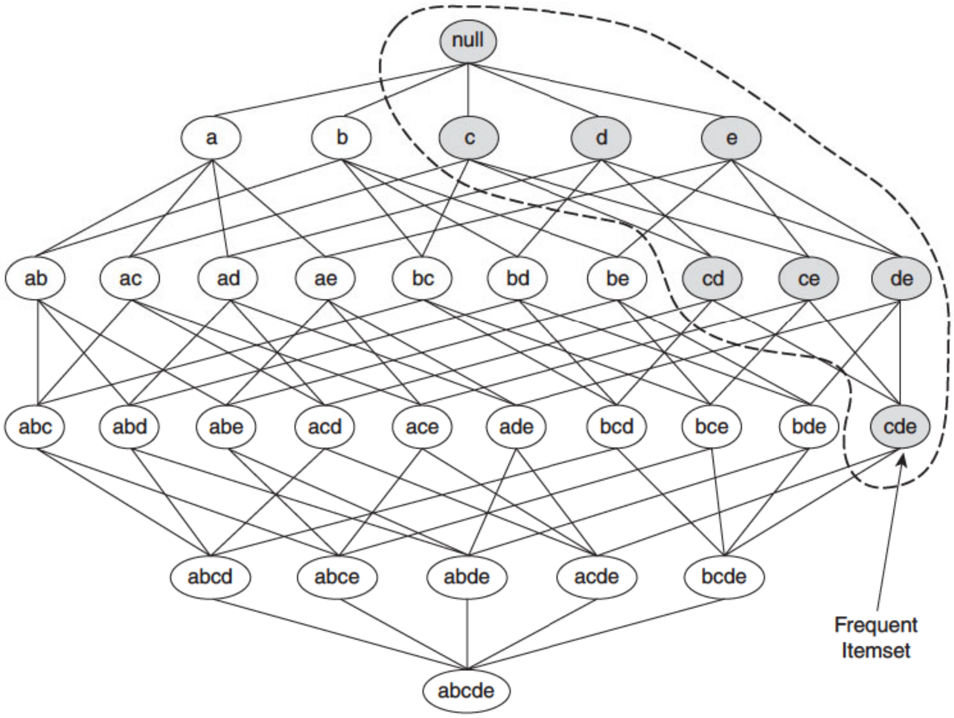


Figure 6.3. An illustration of the *Apriori* principle. If $\{c, d, e\}$ is frequent, then all subsets of this itemset are frequent.

Conversely, if an itemset such as $\{a, b\}$ is infrequent, then all of its supersets must be infrequent too. As illustrated in Figure 6.4, the entire subgraph containing the supersets of $\{a, b\}$ can be pruned immediately once $\{a, b\}$ is found to be infrequent. This strategy of trimming the exponential search space based on the support measure is known as **support-based pruning**. Such a pruning strategy is made possible by a key property of the support measure, namely, that the support for an itemset never exceeds the support for its subsets. This property is also known as the **anti-monotone** property of the support measure.

Definition 6.2 (Monotonicity Property). Let I be a set of items, and $J = 2^I$ be the power set of I. A measure f is monotone (or upward closed) if

$$\forall X, Y \in J: \ (X \subseteq Y) \longrightarrow f(X) \leq f(Y),$$

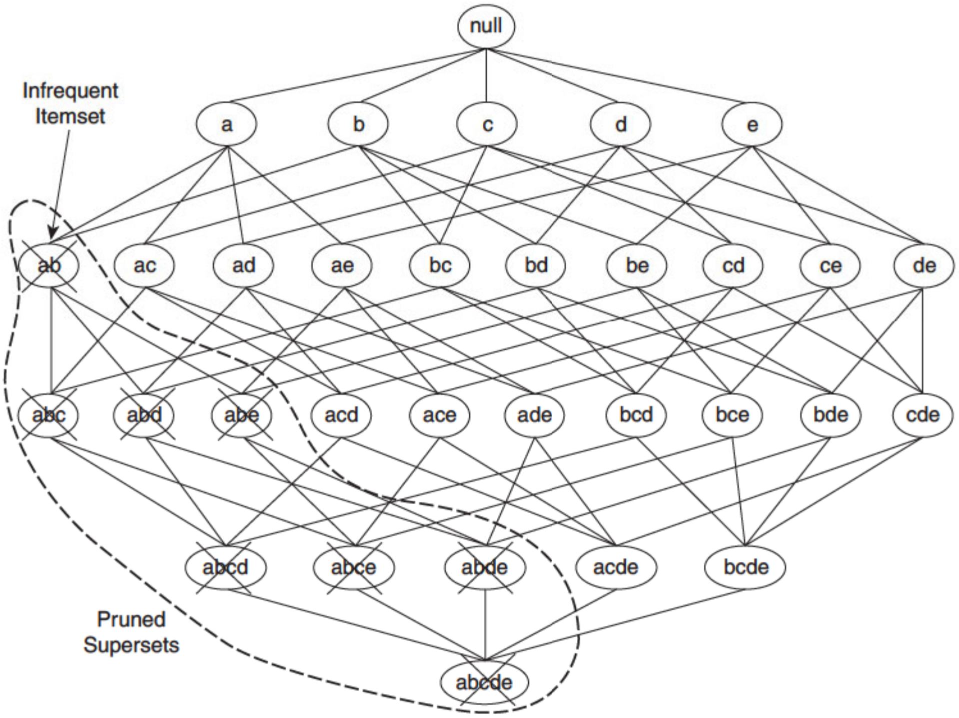


Figure 6.4. An illustration of support-based pruning. If $\{a, b\}$ is infrequent, then all supersets of $\{a, b\}$ are infrequent.

which means that if X is a subset of Y, then $f(X)$ must not exceed $f(Y)$. On the other hand, f is anti-monotone (or downward closed) if

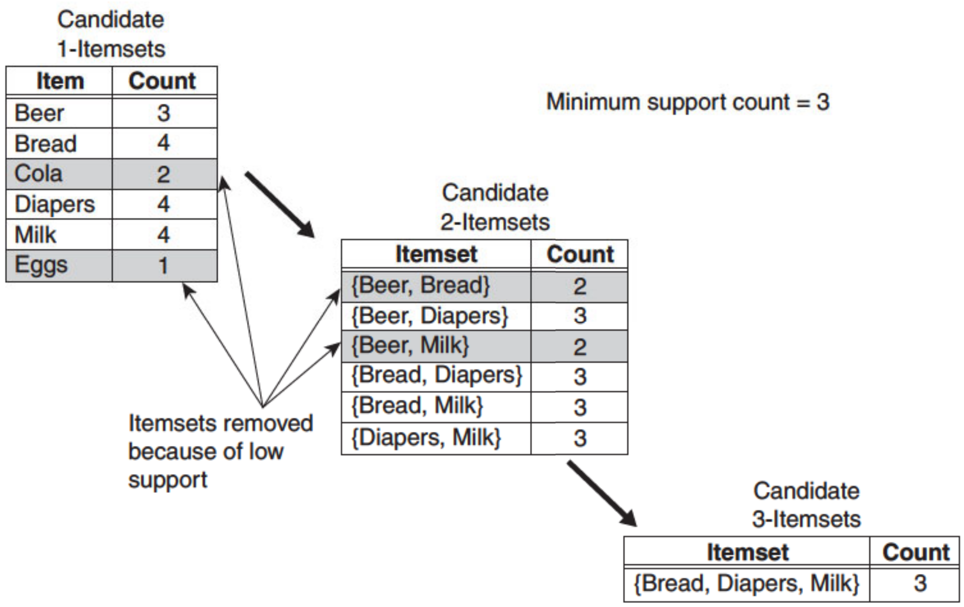
$$\forall X, Y \in J: \ (X \subseteq Y) \longrightarrow f(Y) \leq f(X),$$

which means that if X is a subset of Y, then $f(Y)$ must not exceed $f(X)$.

Any measure that possesses an anti-monotone property can be incorporated directly into the mining algorithm to effectively prune the exponential search space of candidate itemsets, as will be shown in the next section.

6.2.2 Frequent Itemset Generation in the *Apriori* Algorithm

Apriori is the first association rule mining algorithm that pioneered the use of support-based pruning to systematically control the exponential growth of candidate itemsets. Figure 6.5 provides a high-level illustration of the frequent itemset generation part of the *Apriori* algorithm for the transactions shown in



Minimum support count = 3
Itemsets removed because of low support

Candidate 1-Itemsets

Item	Count
Beer	3
Bread	4
Cola	2
Diapers	4
Milk	4
Eggs	1

Candidate 2-Itemsets

Itemset	Count
{Beer, Bread}	2
{Beer, Diapers}	3
{Beer, Milk}	2
{Bread, Diapers}	3
{Bread, Milk}	3
{Diapers, Milk}	3

Candidate 3-Itemsets

Itemset	Count
{Bread, Diapers, Milk}	3

Figure 6.5. Illustration of frequent itemset generation using the *Apriori* algorithm.

Table 6.1. We assume that the support threshold is 60%, which is equivalent to a minimum support count equal to 3.

Initially, every item is considered as a candidate 1-itemset. After counting their supports, the candidate itemsets {Cola} and {Eggs} are discarded because they appear in fewer than three transactions. In the next iteration, candidate 2-itemsets are generated using only the frequent 1-itemsets because the *Apriori* principle ensures that all supersets of the infrequent 1-itemsets must be infrequent. Because there are only four frequent 1-itemsets, the number of candidate 2-itemsets generated by the algorithm is $\binom{4}{2} = 6$. Two of these six candidates, {Beer, Bread} and {Beer, Milk}, are subsequently found to be infrequent after computing their support values. The remaining four candidates are frequent, and thus will be used to generate candidate 3-itemsets. Without support-based pruning, there are $\binom{6}{3} = 20$ candidate 3-itemsets that can be formed using the six items given in this example. With the *Apriori* principle, we only need to keep candidate 3-itemsets whose subsets are frequent. The only candidate that has this property is {Bread, Diapers, Milk}.

The effectiveness of the *Apriori* pruning strategy can be shown by counting the number of candidate itemsets generated. A brute-force strategy of enumerating all itemsets (up to size 3) as candidates will produce

$$\binom{6}{1} + \binom{6}{2} + \binom{6}{3} = 6 + 15 + 20 = 41$$

candidates. With the *Apriori* principle, this number decreases to

$$\binom{6}{1} + \binom{4}{2} + 1 = 6 + 6 + 1 = 13$$

candidates, which represents a 68% reduction in the number of candidate itemsets even in this simple example.

The pseudocode for the frequent itemset generation part of the *Apriori* algorithm is shown in Algorithm 6.1. Let C_k denote the set of candidate k-itemsets and F_k denote the set of frequent k-itemsets:

- The algorithm initially makes a single pass over the data set to determine the support of each item. Upon completion of this step, the set of all frequent 1-itemsets, F_1, will be known (steps 1 and 2).

- Next, the algorithm will iteratively generate new candidate k-itemsets using the frequent $(k-1)$-itemsets found in the previous iteration (step 5). Candidate generation is implemented using a function called apriori-gen, which is described in Section 6.2.3.

Algorithm 6.1 Frequent itemset generation of the *Apriori* algorithm.

```
 1: $k = 1$.
 2: $F_k = \{\ i \mid i \in I \wedge \sigma(\{i\}) \geq N \times minsup\}$.   {Find all frequent 1-itemsets}
 3: repeat
 4:   $k = k + 1$.
 5:   $C_k$ = apriori-gen($F_{k-1}$).   {Generate candidate itemsets}
 6:   for each transaction $t \in T$ do
 7:     $C_t$ = subset($C_k$, $t$).   {Identify all candidates that belong to $t$}
 8:     for each candidate itemset $c \in C_t$ do
 9:       $\sigma(c) = \sigma(c) + 1$.   {Increment support count}
10:     end for
11:   end for
12:   $F_k = \{\ c \mid c \in C_k \wedge \sigma(c) \geq N \times minsup\}$.   {Extract the frequent $k$-itemsets}
13: until $F_k = \emptyset$
14: Result = $\bigcup F_k$.
```

- To count the support of the candidates, the algorithm needs to make an additional pass over the data set (steps 6–10). The subset function is used to determine all the candidate itemsets in C_k that are contained in each transaction t. The implementation of this function is described in Section 6.2.4.

- After counting their supports, the algorithm eliminates all candidate itemsets whose support counts are less than *minsup* (step 12).

- The algorithm terminates when there are no new frequent itemsets generated, i.e., $F_k = \emptyset$ (step 13).

The frequent itemset generation part of the *Apriori* algorithm has two important characteristics. First, it is a **level-wise** algorithm; i.e., it traverses the itemset lattice one level at a time, from frequent 1-itemsets to the maximum size of frequent itemsets. Second, it employs a **generate-and-test** strategy for finding frequent itemsets. At each iteration, new candidate itemsets are generated from the frequent itemsets found in the previous iteration. The support for each candidate is then counted and tested against the *minsup* threshold. The total number of iterations needed by the algorithm is $k_{\max}+1$, where $k_{\max}$ is the maximum size of the frequent itemsets.

6.2.3 Candidate Generation and Pruning

The apriori-gen function shown in Step 5 of Algorithm 6.1 generates candidate itemsets by performing the following two operations:

1. **Candidate Generation.** This operation generates new candidate k-itemsets based on the frequent $(k-1)$-itemsets found in the previous iteration.

2. **Candidate Pruning.** This operation eliminates some of the candidate k-itemsets using the support-based pruning strategy.

To illustrate the candidate pruning operation, consider a candidate k-itemset, $X = \{i_1, i_2, \dots, i_k\}$. The algorithm must determine whether all of its proper subsets, $X - \{i_j\}$ $(\forall j = 1, 2, \dots, k)$, are frequent. If one of them is infrequent, then X is immediately pruned. This approach can effectively reduce the number of candidate itemsets considered during support counting. The complexity of this operation is $O(k)$ for each candidate k-itemset. However, as will be shown later, we do not have to examine all k subsets of a given candidate itemset. If m of the k subsets were used to generate a candidate, we only need to check the remaining $k-m$ subsets during candidate pruning.

In principle, there are many ways to generate candidate itemsets. The following is a list of requirements for an effective candidate generation procedure:

1. It should avoid generating too many unnecessary candidates. A candidate itemset is unnecessary if at least one of its subsets is infrequent. Such a candidate is guaranteed to be infrequent according to the anti-monotone property of support.

2. It must ensure that the candidate set is complete, i.e., no frequent itemsets are left out by the candidate generation procedure. To ensure completeness, the set of candidate itemsets must subsume the set of all frequent itemsets, i.e., $\forall k : F_k \subseteq C_k$.

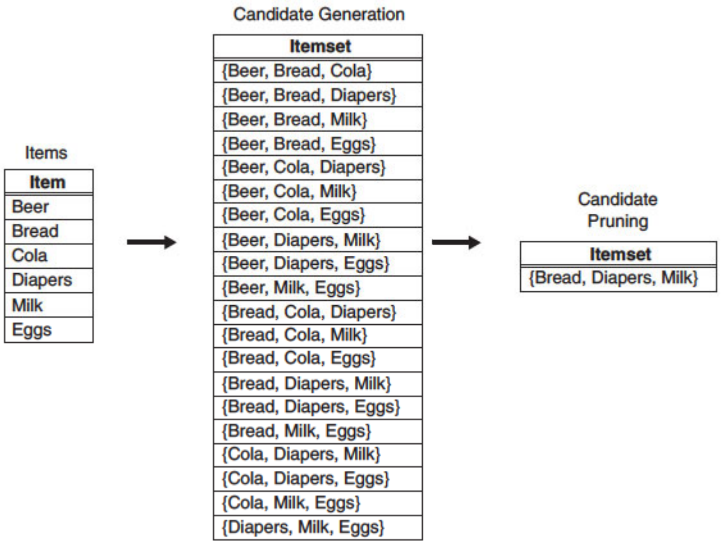
3. It should not generate the same candidate itemset more than once. For example, the candidate itemset $\{a, b, c, d\}$ can be generated in many ways—by merging $\{a, b, c\}$ with $\{d\}$, $\{b, d\}$ with $\{a, c\}$, $\{c\}$ with $\{a, b, d\}$, etc. Generation of duplicate candidates leads to wasted computations and thus should be avoided for efficiency reasons.

Next, we will briefly describe several candidate generation procedures, including the one used by the apriori-gen function.

Brute-Force Method The brute-force method considers every k-itemset as a potential candidate and then applies the candidate pruning step to remove any unnecessary candidates (see Figure 6.6). The number of candidate itemsets generated at level k is equal to $\binom{d}{k}$, where d is the total number of items. Although candidate generation is rather trivial, candidate pruning becomes extremely expensive because a large number of itemsets must be examined. Given that the amount of computations needed for each candidate is $O(k)$, the overall complexity of this method is $O\big(\sum_{k=1}^{d} k \times \binom{d}{k}\big) = O\big(d \cdot 2^{d-1}\big)$.

$\mathbf{F}_{k-1} \times \mathbf{F}_1$ Method An alternative method for candidate generation is to extend each frequent $(k-1)$-itemset with other frequent items. Figure 6.7 illustrates how a frequent 2-itemset such as {`Beer`, `Diapers`} can be augmented with a frequent item such as `Bread` to produce a candidate 3-itemset {`Beer`, `Diapers`, `Bread`}. This method will produce $O(|F_{k-1}| \times |F_1|)$ candidate k-itemsets, where $|F_j|$ is the number of frequent j-itemsets. The overall complexity of this step is $O(\sum_k k|F_{k-1}||F_1|)$.

The procedure is complete because every frequent k-itemset is composed of a frequent $(k-1)$-itemset and a frequent 1-itemset. Therefore, all frequent k-itemsets are part of the candidate k-itemsets generated by this procedure.



Items

Item
Beer
Bread
Cola
Diapers
Milk
Eggs

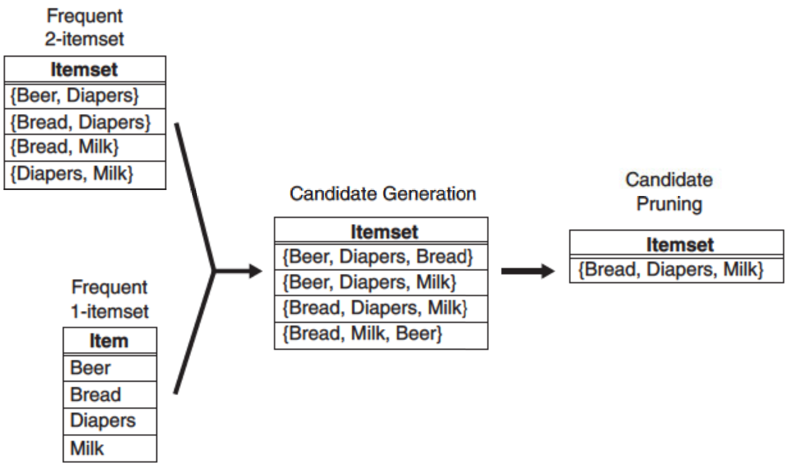
Candidate Generation

Itemset
{Beer, Bread, Cola}
{Beer, Bread, Diapers}
{Beer, Bread, Milk}
{Beer, Bread, Eggs}
{Beer, Cola, Diapers}
{Beer, Cola, Milk}
{Beer, Cola, Eggs}
{Beer, Diapers, Milk}
{Beer, Diapers, Eggs}
{Beer, Milk, Eggs}
{Bread, Cola, Diapers}
{Bread, Cola, Milk}
{Bread, Cola, Eggs}
{Bread, Diapers, Milk}
{Bread, Diapers, Eggs}
{Bread, Milk, Eggs}
{Cola, Diapers, Milk}
{Cola, Diapers, Eggs}
{Cola, Milk, Eggs}
{Diapers, Milk, Eggs}

Candidate Pruning

Itemset
{Bread, Diapers, Milk}

Figure 6.6. A brute-force method for generating candidate 3-itemsets.



Frequent 2-itemset

Itemset
{Beer, Diapers}
{Bread, Diapers}
{Bread, Milk}
{Diapers, Milk}

Frequent 1-itemset

Item
Beer
Bread
Diapers
Milk

Candidate Generation

Itemset
{Beer, Diapers, Bread}
{Beer, Diapers, Milk}
{Bread, Diapers, Milk}
{Bread, Milk, Beer}

Candidate Pruning

Itemset
{Bread, Diapers, Milk}

Figure 6.7. Generating and pruning candidate k-itemsets by merging a frequent $(k-1)$-itemset with a frequent item. Note that some of the candidates are unnecessary because their subsets are infrequent.

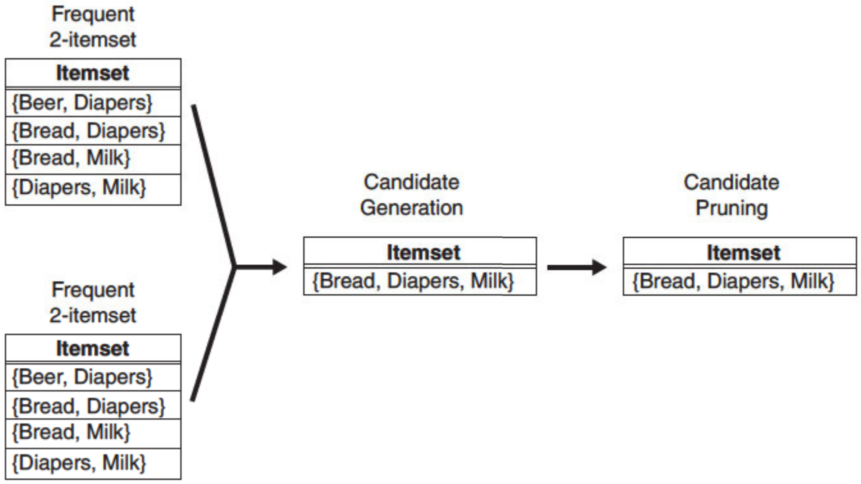
This approach, however, does not prevent the same candidate itemset from being generated more than once. For instance, {`Bread`, `Diapers`, `Milk`} can be generated by merging {`Bread`, `Diapers`} with {`Milk`}, {`Bread`, `Milk`} with {`Diapers`}, or {`Diapers`, `Milk`} with {`Bread`}. One way to avoid generating duplicate candidates is by ensuring that the items in each frequent itemset are kept sorted in their lexicographic order. Each frequent $(k-1)$-itemset X is then extended with frequent items that are lexicographically larger than the items in X. For example, the itemset {Bread, Diapers} can be augmented with {Milk} since Milk is lexicographically larger than Bread and Diapers. However, we should not augment {Diapers, Milk} with {Bread} nor {Bread, Milk} with {Diapers} because they violate the lexicographic ordering condition.

While this procedure is a substantial improvement over the brute-force method, it can still produce a large number of unnecessary candidates. For example, the candidate itemset obtained by merging {Beer, Diapers} with {Milk} is unnecessary because one of its subsets, {Beer, Milk}, is infrequent. There are several heuristics available to reduce the number of unnecessary candidates. For example, note that, for every candidate k-itemset that survives the pruning step, every item in the candidate must be contained in at least $k-1$ of the frequent $(k-1)$-itemsets. Otherwise, the candidate is guaranteed to be infrequent. For example, {Beer, Diapers, Milk} is a viable candidate 3-itemset only if every item in the candidate, including Beer, is contained in at least two frequent 2-itemsets. Since there is only one frequent 2-itemset containing Beer, all candidate itemsets involving Beer must be infrequent.

$\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ Method The candidate generation procedure in the apriori-gen function merges a pair of frequent $(k-1)$-itemsets only if their first $k-2$ items are identical. Let $A = \{a_1, a_2, \ldots, a_{k-1}\}$ and $B = \{b_1, b_2, \ldots, b_{k-1}\}$ be a pair of frequent $(k-1)$-itemsets. A and B are merged if they satisfy the following conditions:

$$a_i = b_i \text{ (for } i = 1, 2, \ldots, k-2) \text{ and } a_{k-1} \neq b_{k-1}.$$

In Figure 6.8, the frequent itemsets {Bread, Diapers} and {Bread, Milk} are merged to form a candidate 3-itemset {Bread, Diapers, Milk}. The algorithm does not have to merge {Beer, Diapers} with {Diapers, Milk} because the first item in both itemsets is different. Indeed, if {Beer, Diapers, Milk} is a viable candidate, it would have been obtained by merging {Beer, Diapers} with {Beer, Milk} instead. This example illustrates both the completeness of the candidate generation procedure and the advantages of using lexicographic ordering to prevent duplicate candidates. However, because each candidate is obtained by merging a pair of frequent $(k-1)$-itemsets, an additional candidate pruning step is needed to ensure that the remaining $k-2$ subsets of the candidate are frequent.



Itemset
{Beer, Diapers}
{Bread, Diapers}
{Bread, Milk}
{Diapers, Milk}

Itemset
{Beer, Diapers}
{Bread, Diapers}
{Bread, Milk}
{Diapers, Milk}

Itemset
{Bread, Diapers, Milk}

Itemset
{Bread, Diapers, Milk}

Figure 6.8. Generating and pruning candidate k-itemsets by merging pairs of frequent $(k-1)$-itemsets.

6.2.4 Support Counting

Support counting is the process of determining the frequency of occurrence for every candidate itemset that survives the candidate pruning step of the apriori-gen function. Support counting is implemented in steps 6 through 11 of Algorithm 6.1. One approach for doing this is to compare each transaction against every candidate itemset (see Figure 6.2) and to update the support counts of candidates contained in the transaction. This approach is computationally expensive, especially when the numbers of transactions and candidate itemsets are large.

An alternative approach is to enumerate the itemsets contained in each transaction and use them to update the support counts of their respective candidate itemsets. To illustrate, consider a transaction t that contains five items, $\{1, 2, 3, 5, 6\}$. There are $\binom{5}{3} = 10$ itemsets of size 3 contained in this transaction. Some of the itemsets may correspond to the candidate 3-itemsets under investigation, in which case, their support counts are incremented. Other subsets of t that do not correspond to any candidates can be ignored.

Figure 6.9 shows a systematic way for enumerating the 3-itemsets contained in t. Assuming that each itemset keeps its items in increasing lexicographic order, an itemset can be enumerated by specifying the smallest item first, followed by the larger items. For instance, given $t = \{1, 2, 3, 5, 6\}$, all the 3-itemsets contained in t must begin with item 1, 2, or 3. It is not possible to construct a 3-itemset that begins with items 5 or 6 because there are only two

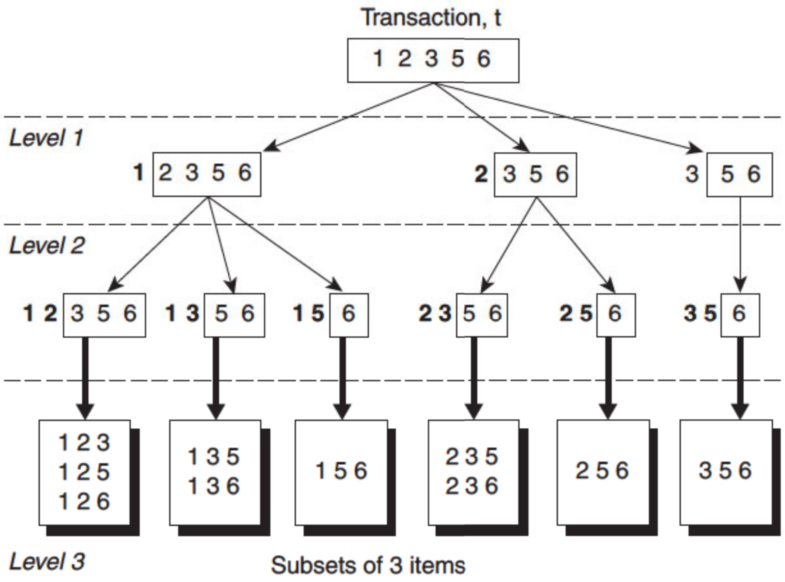
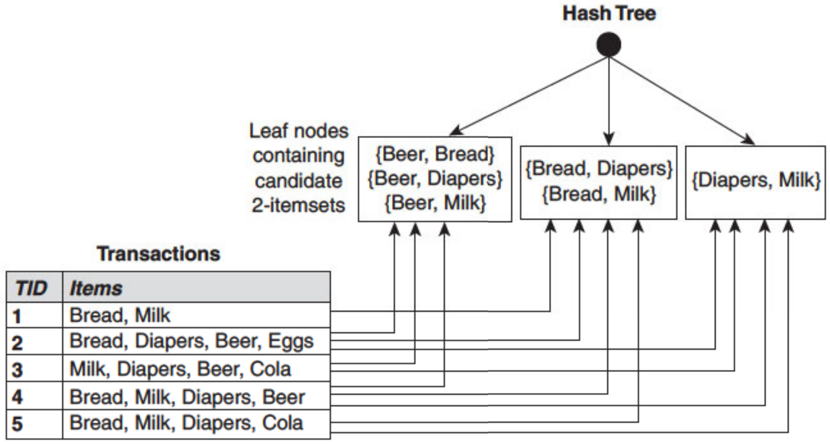


Figure 6.9. Enumerating subsets of three items from a transaction t.

items in t whose labels are greater than or equal to 5. The number of ways to specify the first item of a 3-itemset contained in t is illustrated by the Level 1 prefix structures depicted in Figure 6.9. For instance, $1\ \boxed{2\ 3\ 5\ 6}$ represents a 3-itemset that begins with item 1, followed by two more items chosen from the set $\{2, 3, 5, 6\}$.

After fixing the first item, the prefix structures at Level 2 represent the number of ways to select the second item. For example, $1\ 2\ \boxed{3\ 5\ 6}$ corresponds to itemsets that begin with prefix (1 2) and are followed by items 3, 5, or 6. Finally, the prefix structures at Level 3 represent the complete set of 3-itemsets contained in t. For example, the 3-itemsets that begin with prefix {1 2} are $\{1, 2, 3\}$, $\{1, 2, 5\}$, and $\{1, 2, 6\}$, while those that begin with prefix {2 3} are $\{2, 3, 5\}$ and $\{2, 3, 6\}$.

The prefix structures shown in Figure 6.9 demonstrate how itemsets contained in a transaction can be systematically enumerated, i.e., by specifying their items one by one, from the leftmost item to the rightmost item. We still have to determine whether each enumerated 3-itemset corresponds to an existing candidate itemset. If it matches one of the candidates, then the support count of the corresponding candidate is incremented. In the next section, we illustrate how this matching operation can be performed efficiently using a hash tree structure.



TID	Items
1	Bread, Milk
2	Bread, Diapers, Beer, Eggs
3	Milk, Diapers, Beer, Cola
4	Bread, Milk, Diapers, Beer
5	Bread, Milk, Diapers, Cola

Figure 6.10. Counting the support of itemsets using hash structure.

Support Counting Using a Hash Tree

In the *Apriori* algorithm, candidate itemsets are partitioned into different buckets and stored in a hash tree. During support counting, itemsets contained in each transaction are also hashed into their appropriate buckets. That way, instead of comparing each itemset in the transaction with every candidate itemset, it is matched only against candidate itemsets that belong to the same bucket, as shown in Figure 6.10.

Figure 6.11 shows an example of a hash tree structure. Each internal node of the tree uses the following hash function, $h(p) = p \bmod 3$, to determine which branch of the current node should be followed next. For example, items 1, 4, and 7 are hashed to the same branch (i.e., the leftmost branch) because they have the same remainder after dividing the number by 3. All candidate itemsets are stored at the leaf nodes of the hash tree. The hash tree shown in Figure 6.11 contains 15 candidate 3-itemsets, distributed across 9 leaf nodes.

Consider a transaction, $t = \{1, 2, 3, 5, 6\}$. To update the support counts of the candidate itemsets, the hash tree must be traversed in such a way that all the leaf nodes containing candidate 3-itemsets belonging to t must be visited at least once. Recall that the 3-itemsets contained in t must begin with items 1, 2, or 3, as indicated by the Level 1 prefix structures shown in Figure 6.9. Therefore, at the root node of the hash tree, the items 1, 2, and 3 of the transaction are hashed separately. Item 1 is hashed to the left child of the root node, item 2 is hashed to the middle child, and item 3 is hashed to the right child. At the next level of the tree, the transaction is hashed on the second

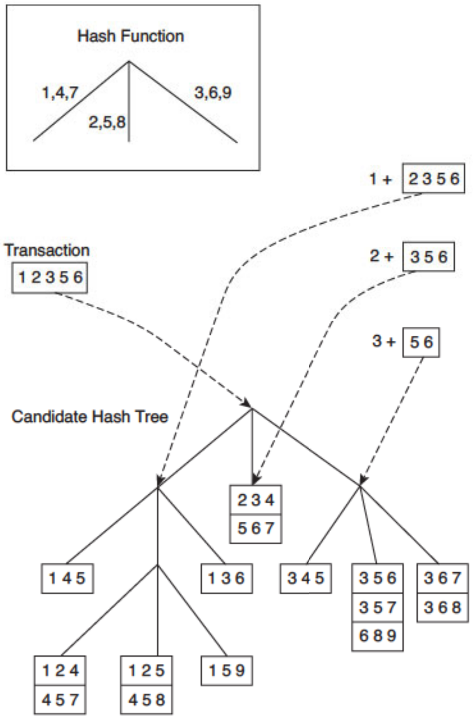


Figure 6.11. Hashing a transaction at the root node of a hash tree.

item listed in the Level 2 structures shown in Figure 6.9. For example, after hashing on item 1 at the root node, items 2, 3, and 5 of the transaction are hashed. Items 2 and 5 are hashed to the middle child, while item 3 is hashed to the right child, as shown in Figure 6.12. This process continues until the leaf nodes of the hash tree are reached. The candidate itemsets stored at the visited leaf nodes are compared against the transaction. If a candidate is a subset of the transaction, its support count is incremented. In this example, 5 out of the 9 leaf nodes are visited and 9 out of the 15 itemsets are compared against the transaction.

6.2.5 Computational Complexity

The computational complexity of the *Apriori* algorithm can be affected by the following factors.

Support Threshold Lowering the support threshold often results in more itemsets being declared as frequent. This has an adverse effect on the com-

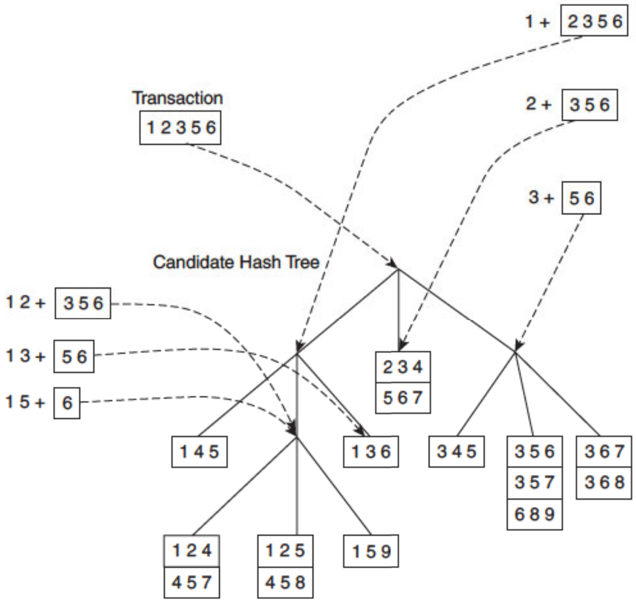


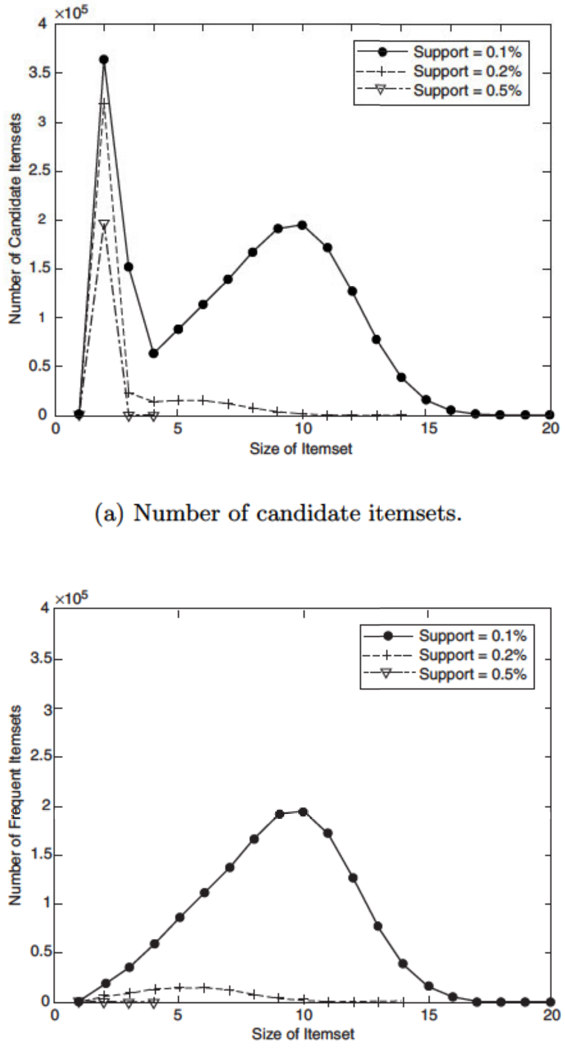
Figure 6.12. Subset operation on the leftmost subtree of the root of a candidate hash tree.

putational complexity of the algorithm because more candidate itemsets must be generated and counted, as shown in Figure 6.13. The maximum size of frequent itemsets also tends to increase with lower support thresholds. As the maximum size of the frequent itemsets increases, the algorithm will need to make more passes over the data set.

Number of Items (Dimensionality) As the number of items increases, more space will be needed to store the support counts of items. If the number of frequent items also grows with the dimensionality of the data, the computation and I/O costs will increase because of the larger number of candidate itemsets generated by the algorithm.

Number of Transactions Since the *Apriori* algorithm makes repeated passes over the data set, its run time increases with a larger number of transactions.

Average Transaction Width For dense data sets, the average transaction width can be very large. This affects the complexity of the *Apriori* algorithm in two ways. First, the maximum size of frequent itemsets tends to increase as the

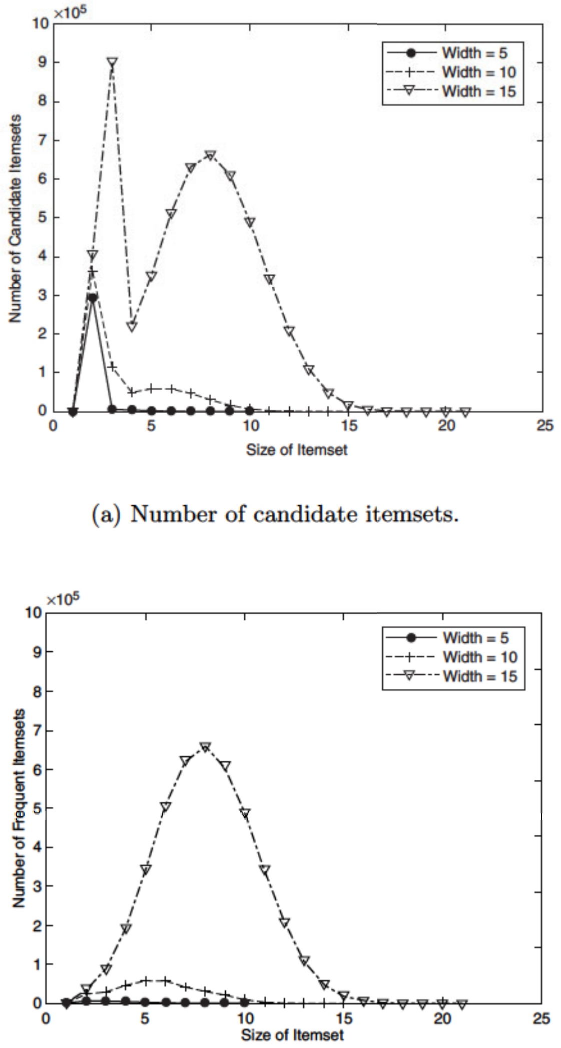


(a) Number of candidate itemsets.

(b) Number of frequent itemsets.

Figure 6.13. Effect of support threshold on the number of candidate and frequent itemsets.

average transaction width increases. As a result, more candidate itemsets must be examined during candidate generation and support counting, as illustrated in Figure 6.14. Second, as the transaction width increases, more itemsets



(a) Number of candidate itemsets.

(b) Number of Frequent Itemsets.

Figure 6.14. Effect of average transaction width on the number of candidate and frequent itemsets.

are contained in the transaction. This will increase the number of hash tree traversals performed during support counting.

A detailed analysis of the time complexity for the *Apriori* algorithm is presented next.

Generation of frequent 1-itemsets For each transaction, we need to update the support count for every item present in the transaction. Assuming that w is the average transaction width, this operation requires $O(Nw)$ time, where N is the total number of transactions.

Candidate generation To generate candidate k-itemsets, pairs of frequent $(k-1)$-itemsets are merged to determine whether they have at least $k-2$ items in common. Each merging operation requires at most $k-2$ equality comparisons. In the best-case scenario, every merging step produces a viable candidate k-itemset. In the worst-case scenario, the algorithm must merge every pair of frequent $(k-1)$-itemsets found in the previous iteration. Therefore, the overall cost of merging frequent itemsets is

$$\sum_{k=2}^{w}(k-2)|C_k| < \text{ Cost of merging } < \sum_{k=2}^{w}(k-2)|F_{k-1}|^2.$$

A hash tree is also constructed during candidate generation to store the candidate itemsets. Because the maximum depth of the tree is k, the cost for populating the hash tree with candidate itemsets is $O\big(\sum_{k=2}^{w} k|C_k|\big)$. During candidate pruning, we need to verify that the $k-2$ subsets of every candidate k-itemset are frequent. Since the cost for looking up a candidate in a hash tree is $O(k)$, the candidate pruning step requires $O\big(\sum_{k=2}^{w} k(k-2)|C_k|\big)$ time.

Support counting Each transaction of length $|t|$ produces $\binom{|t|}{k}$ itemsets of size k. This is also the effective number of hash tree traversals performed for each transaction. The cost for support counting is $O\big(N\sum_k \binom{w}{k}\alpha_k\big)$, where w is the maximum transaction width and α_k is the cost for updating the support count of a candidate k-itemset in the hash tree.

6.3 Rule Generation

This section describes how to extract association rules efficiently from a given frequent itemset. Each frequent k-itemset, Y, can produce up to 2^k-2 association rules, ignoring rules that have empty antecedents or consequents ($\emptyset \longrightarrow Y$ or $Y \longrightarrow \emptyset$). An association rule can be extracted by partitioning the itemset Y into two non-empty subsets, X and $Y-X$, such that $X \longrightarrow Y-X$ satisfies the confidence threshold. Note that all such rules must have already met the support threshold because they are generated from a frequent itemset.

Example 6.2. Let $X = \{1, 2, 3\}$ be a frequent itemset. There are six candidate association rules that can be generated from X: $\{1, 2\} \longrightarrow \{3\}$, $\{1, 3\} \longrightarrow \{2\}$, $\{2, 3\} \longrightarrow \{1\}$, $\{1\} \longrightarrow \{2, 3\}$, $\{2\} \longrightarrow \{1, 3\}$, and $\{3\} \longrightarrow \{1, 2\}$. As each of their support is identical to the support for X, the rules must satisfy the support threshold. ■

Computing the confidence of an association rule does not require additional scans of the transaction data set. Consider the rule $\{1, 2\} \longrightarrow \{3\}$, which is generated from the frequent itemset $X = \{1, 2, 3\}$. The confidence for this rule is $\sigma(\{1, 2, 3\})/\sigma(\{1, 2\})$. Because $\{1, 2, 3\}$ is frequent, the anti-monotone property of support ensures that $\{1, 2\}$ must be frequent, too. Since the support counts for both itemsets were already found during frequent itemset generation, there is no need to read the entire data set again.

6.3.1 Confidence-Based Pruning

Unlike the support measure, confidence does not have any monotone property. For example, the confidence for $X \longrightarrow Y$ can be larger, smaller, or equal to the confidence for another rule $\tilde{X} \longrightarrow \tilde{Y}$, where $\tilde{X} \subseteq X$ and $\tilde{Y} \subseteq Y$ (see Exercise 3 on page 405). Nevertheless, if we compare rules generated from the same frequent itemset Y, the following theorem holds for the confidence measure.

Theorem 6.2. *If a rule $X \longrightarrow Y - X$ does not satisfy the confidence threshold, then any rule $X' \longrightarrow Y - X'$, where X' is a subset of X, must not satisfy the confidence threshold as well.*

To prove this theorem, consider the following two rules: $X' \longrightarrow Y - X'$ and $X \longrightarrow Y - X$, where $X' \subset X$. The confidence of the rules are $\sigma(Y)/\sigma(X')$ and $\sigma(Y)/\sigma(X)$, respectively. Since X' is a subset of X, $\sigma(X') \geq \sigma(X)$. Therefore, the former rule cannot have a higher confidence than the latter rule.

6.3.2 Rule Generation in *Apriori* Algorithm

The *Apriori* algorithm uses a level-wise approach for generating association rules, where each level corresponds to the number of items that belong to the rule consequent. Initially, all the high-confidence rules that have only one item in the rule consequent are extracted. These rules are then used to generate new candidate rules. For example, if $\{acd\} \longrightarrow \{b\}$ and $\{abd\} \longrightarrow \{c\}$ are high-confidence rules, then the candidate rule $\{ad\} \longrightarrow \{bc\}$ is generated by merging the consequents of both rules. Figure 6.15 shows a lattice structure for the association rules generated from the frequent itemset $\{a, b, c, d\}$. If any node in the lattice has low confidence, then according to Theorem 6.2, the

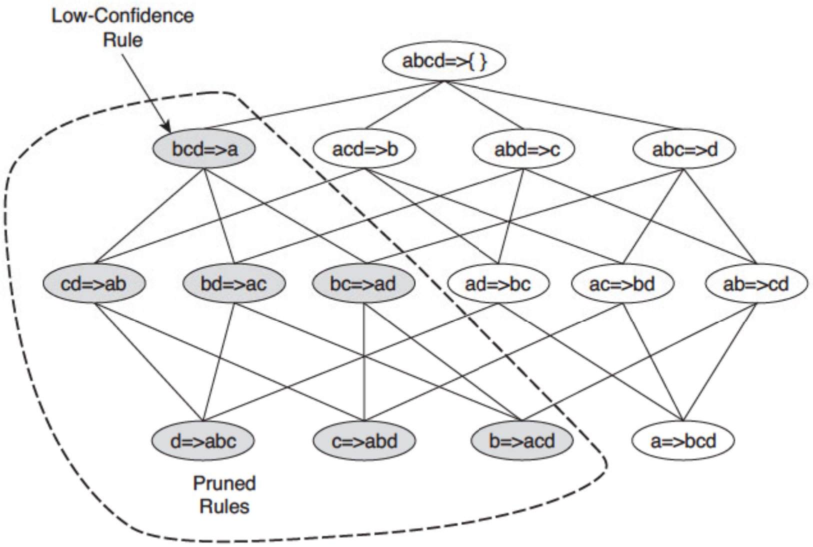


Figure 6.15. Pruning of association rules using the confidence measure.

entire subgraph spanned by the node can be pruned immediately. Suppose the confidence for $\{bcd\} \longrightarrow \{a\}$ is low. All the rules containing item a in its consequent, including $\{cd\} \longrightarrow \{ab\}$, $\{bd\} \longrightarrow \{ac\}$, $\{bc\} \longrightarrow \{ad\}$, and $\{d\} \longrightarrow \{abc\}$ can be discarded.

A pseudocode for the rule generation step is shown in Algorithms 6.2 and 6.3. Note the similarity between the `ap-genrules` procedure given in Algorithm 6.3 and the frequent itemset generation procedure given in Algorithm 6.1. The only difference is that, in rule generation, we do not have to make additional passes over the data set to compute the confidence of the candidate rules. Instead, we determine the confidence of each rule by using the support counts computed during frequent itemset generation.

Algorithm 6.2 Rule generation of the *Apriori* algorithm.

```
for each frequent $k$-itemset $f_k$, $k \geq 2$ do
    $H_1 = \{i \mid i \in f_k\}$        {1-item consequents of the rule.}
    call ap-genrules($f_k, H_1$.)
end for
```

Algorithm 6.3 Procedure `ap-genrules`(f_k, H_m).

```
$k = |f_k|$    {size of frequent itemset.}
$m = |H_m|$    {size of rule consequent.}
if $k > m + 1$ then
  $H_{m+1} =$ apriori-gen($H_m$).
  for each $h_{m+1} \in H_{m+1}$ do
    $conf = \sigma(f_k)/\sigma(f_k - h_{m+1})$.
    if $conf \geq minconf$ then
      output the rule $(f_k - h_{m+1}) \longrightarrow h_{m+1}$.
    else
      delete $h_{m+1}$ from $H_{m+1}$.
    end if
  end for
  call ap-genrules($f_k, H_{m+1}$.)
end if
```

6.3.3 An Example: Congressional Voting Records

This section demonstrates the results of applying association analysis to the voting records of members of the United States House of Representatives. The data is obtained from the 1984 Congressional Voting Records Database, which is available at the UCI machine learning data repository. Each transaction contains information about the party affiliation for a representative along with his or her voting record on 16 key issues. There are 435 transactions and 34 items in the data set. The set of items are listed in Table 6.3.

The *Apriori* algorithm is then applied to the data set with $minsup = 30\%$ and $minconf = 90\%$. Some of the high-confidence rules extracted by the algorithm are shown in Table 6.4. The first two rules suggest that most of the members who voted yes for aid to El Salvador and no for budget resolution and MX missile are Republicans; while those who voted no for aid to El Salvador and yes for budget resolution and MX missile are Democrats. These high-confidence rules show the key issues that divide members from both political parties. If $minconf$ is reduced, we may find rules that contain issues that cut across the party lines. For example, with $minconf = 40\%$, the rules suggest that corporation cutbacks is an issue that receives almost equal number of votes from both parties—52.3% of the members who voted no are Republicans, while the remaining 47.7% of them who voted no are Democrats.

Table 6.3. List of binary attributes from the 1984 United States Congressional Voting Records. Source: The UCI machine learning repository.

1. Republican
2. Democrat
3. handicapped-infants = yes
4. handicapped-infants = no
5. water project cost sharing = yes
6. water project cost sharing = no
7. budget-resolution = yes
8. budget-resolution = no
9. physician fee freeze = yes
10. physician fee freeze = no
11. aid to El Salvador = yes
12. aid to El Salvador = no
13. religious groups in schools = yes
14. religious groups in schools = no
15. anti-satellite test ban = yes
16. anti-satellite test ban = no
17. aid to Nicaragua = yes
18. aid to Nicaragua = no
19. MX-missile = yes
20. MX-missile = no
21. immigration = yes
22. immigration = no
23. synfuel corporation cutback = yes
24. synfuel corporation cutback = no
25. education spending = yes
26. education spending = no
27. right-to-sue = yes
28. right-to-sue = no
29. crime = yes
30. crime = no
31. duty-free-exports = yes
32. duty-free-exports = no
33. export administration act = yes
34. export administration act = no

Table 6.4. Association rules extracted from the 1984 United States Congressional Voting Records.

Association Rule	Confidence
{budget resolution = no, MX-missile=no, aid to El Salvador = yes } ⟶ {Republican}	91.0%
{budget resolution = yes, MX-missile=yes, aid to El Salvador = no } ⟶ {Democrat}	97.5%
{crime = yes, right-to-sue = yes, physician fee freeze = yes} ⟶ {Republican}	93.5%
{crime = no, right-to-sue = no, physician fee freeze = no} ⟶ {Democrat}	100%

6.4 Compact Representation of Frequent Itemsets

In practice, the number of frequent itemsets produced from a transaction data set can be very large. It is useful to identify a small representative set of itemsets from which all other frequent itemsets can be derived. Two such representations are presented in this section in the form of maximal and closed frequent itemsets.

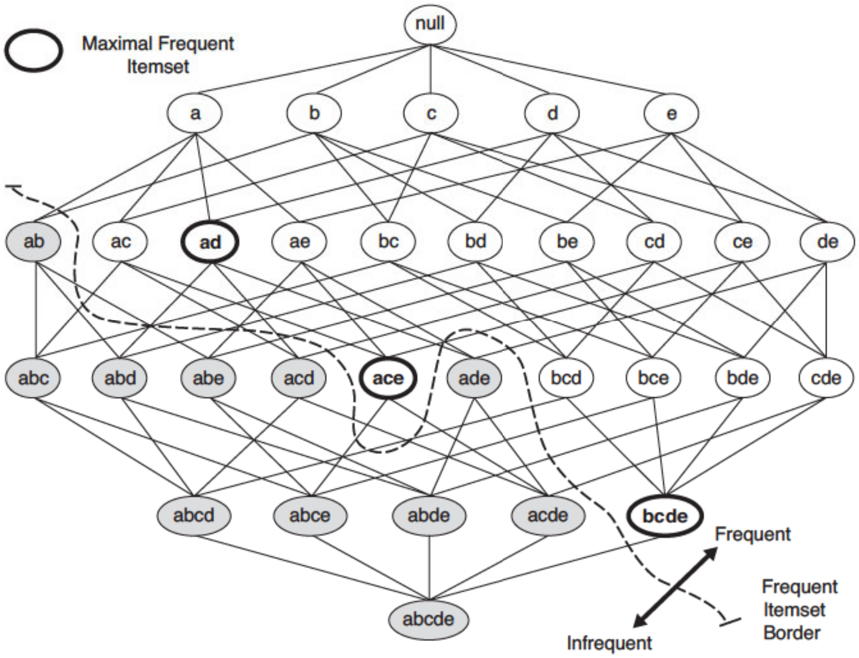


Figure 6.16. Maximal frequent itemset.

6.4.1 Maximal Frequent Itemsets

Definition 6.3 (Maximal Frequent Itemset). A maximal frequent itemset is defined as a frequent itemset for which none of its immediate supersets are frequent.

To illustrate this concept, consider the itemset lattice shown in Figure 6.16. The itemsets in the lattice are divided into two groups: those that are frequent and those that are infrequent. A frequent itemset border, which is represented by a dashed line, is also illustrated in the diagram. Every itemset located above the border is frequent, while those located below the border (the shaded nodes) are infrequent. Among the itemsets residing near the border, $\{a, d\}$, $\{a, c, e\}$, and $\{b, c, d, e\}$ are considered to be maximal frequent itemsets because their immediate supersets are infrequent. An itemset such as $\{a, d\}$ is maximal frequent because all of its immediate supersets, $\{a, b, d\}$, $\{a, c, d\}$, and $\{a, d, e\}$, are infrequent. In contrast, $\{a, c\}$ is non-maximal because one of its immediate supersets, $\{a, c, e\}$, is frequent.

Maximal frequent itemsets effectively provide a compact representation of frequent itemsets. In other words, they form the smallest set of itemsets from which all frequent itemsets can be derived. For example, the frequent itemsets shown in Figure 6.16 can be divided into two groups:

- Frequent itemsets that begin with item a and that may contain items c, d, or e. This group includes itemsets such as $\{a\}$, $\{a, c\}$, $\{a, d\}$, $\{a, e\}$, and $\{a, c, e\}$.
- Frequent itemsets that begin with items b, c, d, or e. This group includes itemsets such as $\{b\}$, $\{b, c\}$, $\{c, d\}$,$\{b, c, d, e\}$, etc.

Frequent itemsets that belong in the first group are subsets of either $\{a, c, e\}$ or $\{a, d\}$, while those that belong in the second group are subsets of $\{b, c, d, e\}$. Hence, the maximal frequent itemsets $\{a, c, e\}$, $\{a, d\}$, and $\{b, c, d, e\}$ provide a compact representation of the frequent itemsets shown in Figure 6.16.

Maximal frequent itemsets provide a valuable representation for data sets that can produce very long, frequent itemsets, as there are exponentially many frequent itemsets in such data. Nevertheless, this approach is practical only if an efficient algorithm exists to explicitly find the maximal frequent itemsets without having to enumerate all their subsets. We briefly describe one such approach in Section 6.5.

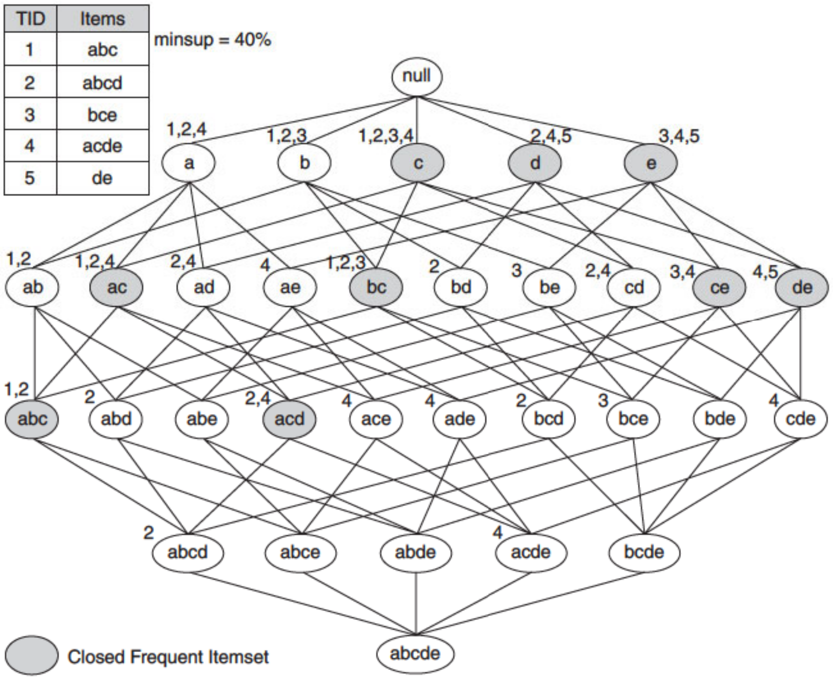
Despite providing a compact representation, maximal frequent itemsets do not contain the support information of their subsets. For example, the support of the maximal frequent itemsets $\{a, c, e\}$, $\{a, d\}$, and $\{b,c,d,e\}$ do not provide any hint about the support of their subsets. An additional pass over the data set is therefore needed to determine the support counts of the non-maximal frequent itemsets. In some cases, it might be desirable to have a minimal representation of frequent itemsets that preserves the support information. We illustrate such a representation in the next section.

6.4.2 Closed Frequent Itemsets

Closed itemsets provide a minimal representation of itemsets without losing their support information. A formal definition of a closed itemset is presented below.

Definition 6.4 (Closed Itemset). An itemset X is closed if none of its immediate supersets has exactly the same support count as X.

Put another way, X is not closed if at least one of its immediate supersets has the same support count as X. Examples of closed itemsets are shown in Figure 6.17. To better illustrate the support count of each itemset, we have associated each node (itemset) in the lattice with a list of its corresponding



TID	Items
1	abc
2	abcd
3	bce
4	acde
5	de

Figure 6.17. An example of the closed frequent itemsets (with minimum support count equal to 40%).

transaction IDs. For example, since the node $\{b, c\}$ is associated with transaction IDs 1, 2, and 3, its support count is equal to three. From the transactions given in this diagram, notice that every transaction that contains b also contains c. Consequently, the support for $\{b\}$ is identical to $\{b, c\}$ and $\{b\}$ should not be considered a closed itemset. Similarly, since c occurs in every transaction that contains both a and d, the itemset $\{a, d\}$ is not closed. On the other hand, $\{b, c\}$ is a closed itemset because it does not have the same support count as any of its supersets.

Definition 6.5 (Closed Frequent Itemset). An itemset is a closed frequent itemset if it is closed and its support is greater than or equal to *minsup*.

In the previous example, assuming that the support threshold is 40%, {b,c} is a closed frequent itemset because its support is 60%. The rest of the closed frequent itemsets are indicated by the shaded nodes.

Algorithms are available to explicitly extract closed frequent itemsets from a given data set. Interested readers may refer to the bibliographic notes at the end of this chapter for further discussions of these algorithms. We can use the closed frequent itemsets to determine the support counts for the non-closed

Algorithm 6.4 Support counting using closed frequent itemsets.

```
Let $C$ denote the set of closed frequent itemsets
Let $k_{\max}$ denote the maximum size of closed frequent itemsets
$F_{k_{\max}} = \{f | f \in C,\ |f| = k_{\max}\}$    {Find all frequent itemsets of size $k_{\max}$.}
for $k = k_{\max} - 1$ downto 1 do
  $F_k = \{f | f \subset F_{k+1},\ |f| = k\}$    {Find all frequent itemsets of size $k$.}
  for each $f \in F_k$ do
    if $f \notin C$ then
      $f.support = \max\{f'.support | f' \in F_{k+1},\ f \subset f'\}$
    end if
  end for
end for
```

frequent itemsets. For example, consider the frequent itemset $\{a, d\}$ shown in Figure 6.17. Because the itemset is not closed, its support count must be identical to one of its immediate supersets. The key is to determine which superset (among $\{a, b, d\}$, $\{a, c, d\}$, or $\{a, d, e\}$) has exactly the same support count as $\{a, d\}$. The *Apriori* principle states that any transaction that contains the superset of $\{a, d\}$ must also contain $\{a, d\}$. However, any transaction that contains $\{a, d\}$ does not have to contain the supersets of $\{a, d\}$. For this reason, the support for $\{a, d\}$ must be equal to the largest support among its supersets. Since $\{a, c, d\}$ has a larger support than both $\{a, b, d\}$ and $\{a, d, e\}$, the support for $\{a, d\}$ must be identical to the support for $\{a, c, d\}$. Using this methodology, an algorithm can be developed to compute the support for the non-closed frequent itemsets. The pseudocode for this algorithm is shown in Algorithm 6.4. The algorithm proceeds in a specific-to-general fashion, i.e., from the largest to the smallest frequent itemsets. This is because, in order to find the support for a non-closed frequent itemset, the support for all of its supersets must be known.

To illustrate the advantage of using closed frequent itemsets, consider the data set shown in Table 6.5, which contains ten transactions and fifteen items. The items can be divided into three groups: (1) Group A, which contains items a_1 through a_5; (2) Group B, which contains items b_1 through b_5; and (3) Group C, which contains items c_1 through c_5. Note that items within each group are perfectly associated with each other and they do not appear with items from another group. Assuming the support threshold is 20%, the total number of frequent itemsets is $3 \times (2^5 - 1) = 93$. However, there are only three closed frequent itemsets in the data: ($\{a_1, a_2, a_3, a_4, a_5\}$, $\{b_1, b_2, b_3, b_4, b_5\}$, and $\{c_1, c_2, c_3, c_4, c_5\}$). It is often sufficient to present only the closed frequent itemsets to the analysts instead of the entire set of frequent itemsets.

Table 6.5. A transaction data set for mining closed itemsets.

TID	a_1	a_2	a_3	a_4	a_5	b_1	b_2	b_3	b_4	b_5	c_1	c_2	c_3	c_4	c_5
1	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
2	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
3	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0
5	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0
6	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**
8	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**
9	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**
10	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**

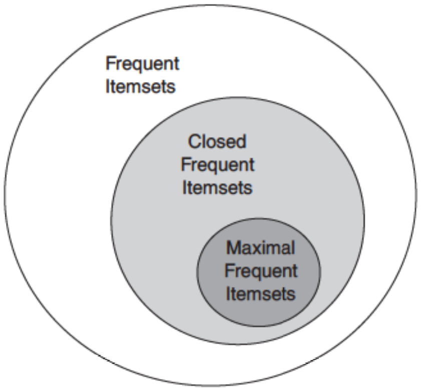


Figure 6.18. Relationships among frequent, maximal frequent, and closed frequent itemsets.

Closed frequent itemsets are useful for removing some of the redundant association rules. An association rule $X \longrightarrow Y$ is redundant if there exists another rule $X' \longrightarrow Y'$, where X is a subset of X' and Y is a subset of Y', such that the support and confidence for both rules are identical. In the example shown in Figure 6.17, $\{b\}$ is not a closed frequent itemset while $\{b, c\}$ is closed. The association rule $\{b\} \longrightarrow \{d, e\}$ is therefore redundant because it has the same support and confidence as $\{b, c\} \longrightarrow \{d, e\}$. Such redundant rules are not generated if closed frequent itemsets are used for rule generation.

Finally, note that all maximal frequent itemsets are closed because none of the maximal frequent itemsets can have the same support count as their immediate supersets. The relationships among frequent, maximal frequent, and closed frequent itemsets are shown in Figure 6.18.

6.5 Alternative Methods for Generating Frequent Itemsets

Apriori is one of the earliest algorithms to have successfully addressed the combinatorial explosion of frequent itemset generation. It achieves this by applying the *Apriori* principle to prune the exponential search space. Despite its significant performance improvement, the algorithm still incurs considerable I/O overhead since it requires making several passes over the transaction data set. In addition, as noted in Section 6.2.5, the performance of the *Apriori* algorithm may degrade significantly for dense data sets because of the increasing width of transactions. Several alternative methods have been developed to overcome these limitations and improve upon the efficiency of the *Apriori* algorithm. The following is a high-level description of these methods.

Traversal of Itemset Lattice A search for frequent itemsets can be conceptually viewed as a traversal on the itemset lattice shown in Figure 6.1. The search strategy employed by an algorithm dictates how the lattice structure is traversed during the frequent itemset generation process. Some search strategies are better than others, depending on the configuration of frequent itemsets in the lattice. An overview of these strategies is presented next.

- **General-to-Specific versus Specific-to-General:** The *Apriori* algorithm uses a general-to-specific search strategy, where pairs of frequent $(k-1)$-itemsets are merged to obtain candidate k-itemsets. This general-to-specific search strategy is effective, provided the maximum length of a frequent itemset is not too long. The configuration of frequent itemsets that works best with this strategy is shown in Figure 6.19(a), where the darker nodes represent infrequent itemsets. Alternatively, a specific-to-general search strategy looks for more specific frequent itemsets first, before finding the more general frequent itemsets. This strategy is useful to discover maximal frequent itemsets in dense transactions, where the frequent itemset border is located near the bottom of the lattice, as shown in Figure 6.19(b). The *Apriori* principle can be applied to prune all subsets of maximal frequent itemsets. Specifically, if a candidate k-itemset is maximal frequent, we do not have to examine any of its subsets of size $k-1$. However, if the candidate k-itemset is infrequent, we need to check all of its $k-1$ subsets in the next iteration. Another approach is to combine both general-to-specific and specific-to-general search strategies. This bidirectional approach requires more space to

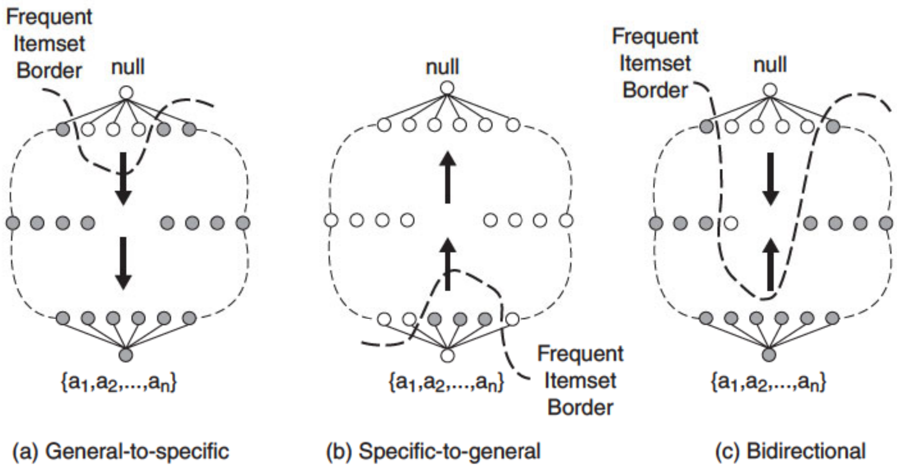
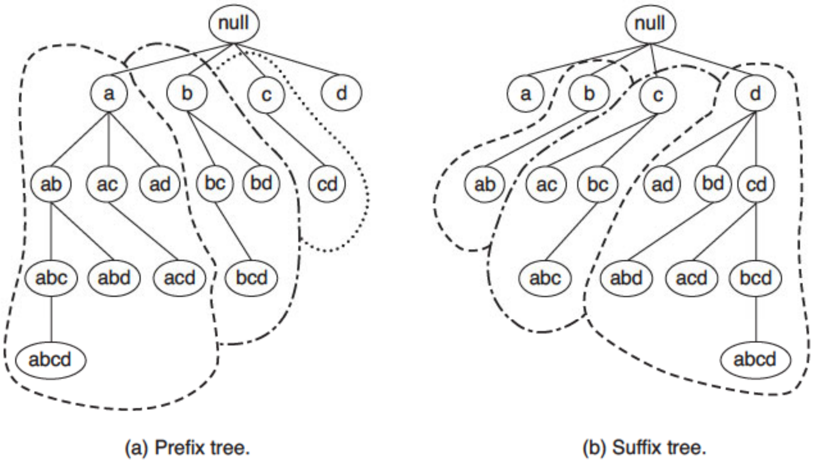


Figure 6.19. General-to-specific, specific-to-general, and bidirectional search.

store the candidate itemsets, but it can help to rapidly identify the frequent itemset border, given the configuration shown in Figure 6.19(c).

- **Equivalence Classes:** Another way to envision the traversal is to first partition the lattice into disjoint groups of nodes (or equivalence classes). A frequent itemset generation algorithm searches for frequent itemsets within a particular equivalence class first before moving to another equivalence class. As an example, the level-wise strategy used in the *Apriori* algorithm can be considered to be partitioning the lattice on the basis of itemset sizes; i.e., the algorithm discovers all frequent 1-itemsets first before proceeding to larger-sized itemsets. Equivalence classes can also be defined according to the prefix or suffix labels of an itemset. In this case, two itemsets belong to the same equivalence class if they share a common prefix or suffix of length k. In the prefix-based approach, the algorithm can search for frequent itemsets starting with the prefix a before looking for those starting with prefixes b, c, and so on. Both prefix-based and suffix-based equivalence classes can be demonstrated using the tree-like structure shown in Figure 6.20.

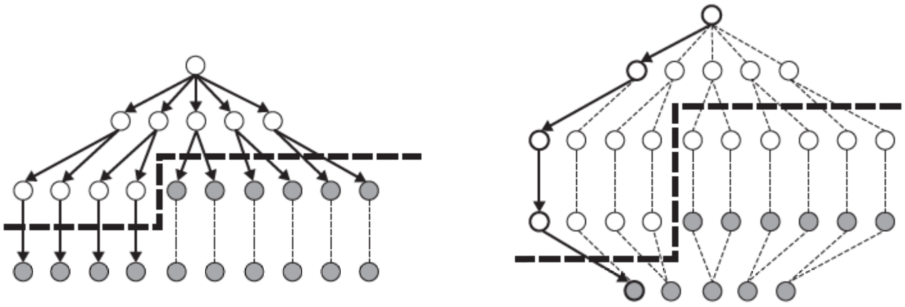
- **Breadth-First versus Depth-First:** The *Apriori* algorithm traverses the lattice in a breadth-first manner, as shown in Figure 6.21(a). It first discovers all the frequent 1-itemsets, followed by the frequent 2-itemsets, and so on, until no new frequent itemsets are generated. The itemset



(a) Prefix tree.

(b) Suffix tree.

Figure 6.20. Equivalence classes based on the prefix and suffix labels of itemsets.



(a) Breadth first

(b) Depth first

Figure 6.21. Breadth-first and depth-first traversals.

lattice can also be traversed in a depth-first manner, as shown in Figures 6.21(b) and 6.22. The algorithm can start from, say, node *a* in Figure 6.22, and count its support to determine whether it is frequent. If so, the algorithm progressively expands the next level of nodes, i.e., *ab*, *abc*, and so on, until an infrequent node is reached, say, *abcd*. It then backtracks to another branch, say, *abce*, and continues the search from there.

The depth-first approach is often used by algorithms designed to find maximal frequent itemsets. This approach allows the frequent itemset border to be detected more quickly than using a breadth-first approach. Once a maximal frequent itemset is found, substantial pruning can be

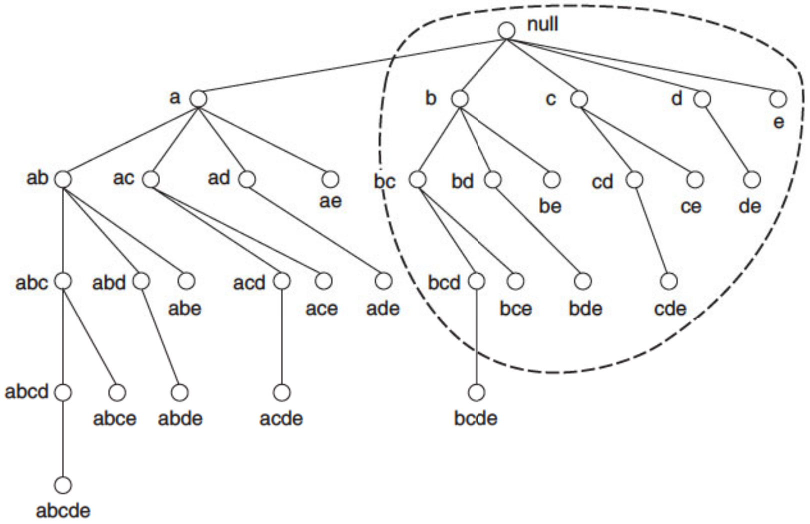


Figure 6.22. Generating candidate itemsets using the depth-first approach.

performed on its subsets. For example, if the node *bcde* shown in Figure 6.22 is maximal frequent, then the algorithm does not have to visit the subtrees rooted at *bd*, *be*, *c*, *d*, and *e* because they will not contain any maximal frequent itemsets. However, if *abc* is maximal frequent, only the nodes such as *ac* and *bc* are not maximal frequent (but the subtrees of *ac* and *bc* may still contain maximal frequent itemsets). The depth-first approach also allows a different kind of pruning based on the support of itemsets. For example, suppose the support for $\{a, b, c\}$ is identical to the support for $\{a, b\}$. The subtrees rooted at *abd* and *abe* can be skipped because they are guaranteed not to have any maximal frequent itemsets. The proof of this is left as an exercise to the readers.

Representation of Transaction Data Set There are many ways to represent a transaction data set. The choice of representation can affect the I/O costs incurred when computing the support of candidate itemsets. Figure 6.23 shows two different ways of representing market basket transactions. The representation on the left is called a **horizontal** data layout, which is adopted by many association rule mining algorithms, including *Apriori*. Another possibility is to store the list of transaction identifiers (TID-list) associated with each item. Such a representation is known as the **vertical** data layout. The support for each candidate itemset is obtained by intersecting the TID-lists of its subset items. The length of the TID-lists shrinks as we progress to larger

Horizontal Data Layout

TID	Items
1	a,b,e
2	b,c,d
3	c,e
4	a,c,d
5	a,b,c,d
6	a,e
7	a,b
8	a,b,c
9	a,c,d
10	b

Vertical Data Layout

a	b	c	d	e
1	1	2	2	1
4	2	3	4	3
5	5	4	5	6
6	7	8	9	
7	8	9		
8	10			
9				

Figure 6.23. Horizontal and vertical data format.

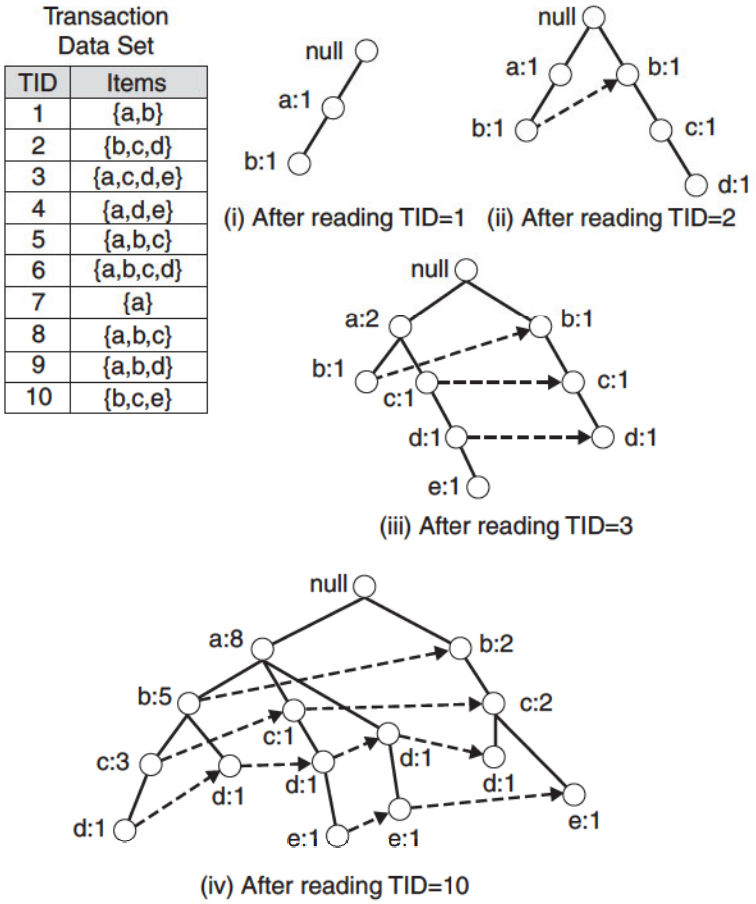
sized itemsets. However, one problem with this approach is that the initial set of TID-lists may be too large to fit into main memory, thus requiring more sophisticated techniques to compress the TID-lists. We describe another effective approach to represent the data in the next section.

6.6 FP-Growth Algorithm

This section presents an alternative algorithm called **FP-growth** that takes a radically different approach to discovering frequent itemsets. The algorithm does not subscribe to the generate-and-test paradigm of *Apriori*. Instead, it encodes the data set using a compact data structure called an **FP-tree** and extracts frequent itemsets directly from this structure. The details of this approach are presented next.

6.6.1 FP-Tree Representation

An FP-tree is a compressed representation of the input data. It is constructed by reading the data set one transaction at a time and mapping each transaction onto a path in the FP-tree. As different transactions can have several items in common, their paths may overlap. The more the paths overlap with one another, the more compression we can achieve using the FP-tree structure. If the size of the FP-tree is small enough to fit into main memory, this will allow us to extract frequent itemsets directly from the structure in memory instead of making repeated passes over the data stored on disk.



Transaction Data Set

TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}

(i) After reading TID=1 (ii) After reading TID=2

(iii) After reading TID=3

(iv) After reading TID=10

Figure 6.24. Construction of an FP-tree.

Figure 6.24 shows a data set that contains ten transactions and five items. The structures of the FP-tree after reading the first three transactions are also depicted in the diagram. Each node in the tree contains the label of an item along with a counter that shows the number of transactions mapped onto the given path. Initially, the FP-tree contains only the root node represented by the `null` symbol. The FP-tree is subsequently extended in the following way:

1. The data set is scanned once to determine the support count of each item. Infrequent items are discarded, while the frequent items are sorted in decreasing support counts. For the data set shown in Figure 6.24, a is the most frequent item, followed by b, c, d, and e.

2. The algorithm makes a second pass over the data to construct the FP-tree. After reading the first transaction, $\{a, b\}$, the nodes labeled as a and b are created. A path is then formed from `null` → a → b to encode the transaction. Every node along the path has a frequency count of 1.

3. After reading the second transaction, $\{b,c,d\}$, a new set of nodes is created for items b, c, and d. A path is then formed to represent the transaction by connecting the nodes `null` → b → c → d. Every node along this path also has a frequency count equal to one. Although the first two transactions have an item in common, which is b, their paths are disjoint because the transactions do not share a common prefix.

4. The third transaction, $\{a,c,d,e\}$, shares a common prefix item (which is a) with the first transaction. As a result, the path for the third transaction, `null` → a → c → d → e, overlaps with the path for the first transaction, `null` → a → b. Because of their overlapping path, the frequency count for node a is incremented to two, while the frequency counts for the newly created nodes, c, d, and e, are equal to one.

5. This process continues until every transaction has been mapped onto one of the paths given in the FP-tree. The resulting FP-tree after reading all the transactions is shown at the bottom of Figure 6.24.

The size of an FP-tree is typically smaller than the size of the uncompressed data because many transactions in market basket data often share a few items in common. In the best-case scenario, where all the transactions have the same set of items, the FP-tree contains only a single branch of nodes. The worst-case scenario happens when every transaction has a unique set of items. As none of the transactions have any items in common, the size of the FP-tree is effectively the same as the size of the original data. However, the physical storage requirement for the FP-tree is higher because it requires additional space to store pointers between nodes and counters for each item.

The size of an FP-tree also depends on how the items are ordered. If the ordering scheme in the preceding example is reversed, i.e., from lowest to highest support item, the resulting FP-tree is shown in Figure 6.25. The tree appears to be denser because the branching factor at the root node has increased from 2 to 5 and the number of nodes containing the high support items such as a and b has increased from 3 to 12. Nevertheless, ordering by decreasing support counts does not always lead to the smallest tree. For example, suppose we augment the data set given in Figure 6.24 with 100 transactions that contain $\{e\}$, 80 transactions that contain $\{d\}$, 60 transactions

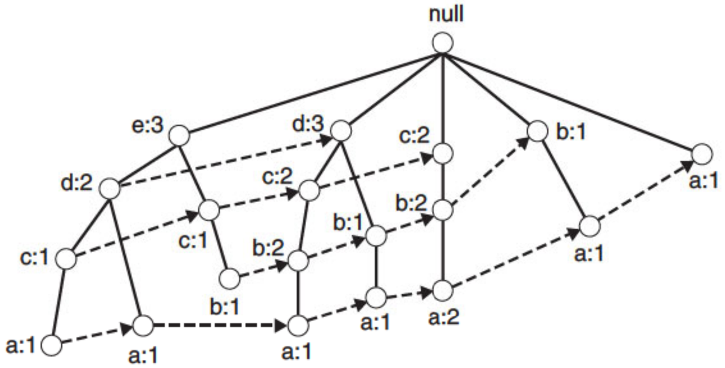


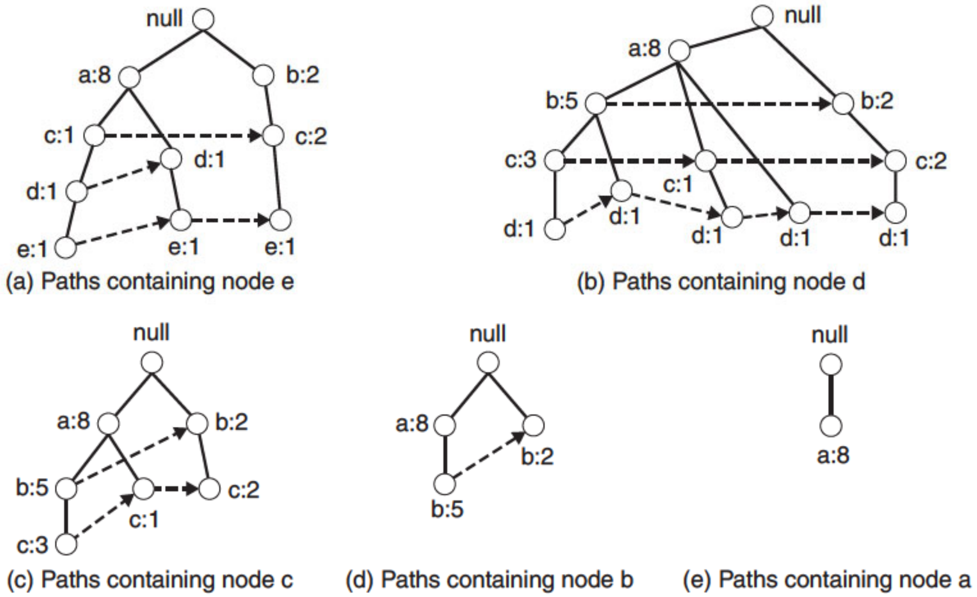
Figure 6.25. An FP-tree representation for the data set shown in Figure 6.24 with a different item ordering scheme.

that contain $\{c\}$, and 40 transactions that contain $\{b\}$. Item e is now most frequent, followed by d, c, b, and a. With the augmented transactions, ordering by decreasing support counts will result in an FP-tree similar to Figure 6.25, while a scheme based on increasing support counts produces a smaller FP-tree similar to Figure 6.24(iv).

An FP-tree also contains a list of pointers connecting between nodes that have the same items. These pointers, represented as dashed lines in Figures 6.24 and 6.25, help to facilitate the rapid access of individual items in the tree. We explain how to use the FP-tree and its corresponding pointers for frequent itemset generation in the next section.

6.6.2 Frequent Itemset Generation in FP-Growth Algorithm

FP-growth is an algorithm that generates frequent itemsets from an FP-tree by exploring the tree in a bottom-up fashion. Given the example tree shown in Figure 6.24, the algorithm looks for frequent itemsets ending in e first, followed by d, c, b, and finally, a. This bottom-up strategy for finding frequent itemsets ending with a particular item is equivalent to the suffix-based approach described in Section 6.5. Since every transaction is mapped onto a path in the FP-tree, we can derive the frequent itemsets ending with a particular item, say, e, by examining only the paths containing node e. These paths can be accessed rapidly using the pointers associated with node e. The extracted paths are shown in Figure 6.26(a). The details on how to process the paths to obtain frequent itemsets will be explained later.



(a) Paths containing node e

(b) Paths containing node d

(c) Paths containing node c

(d) Paths containing node b

(e) Paths containing node a

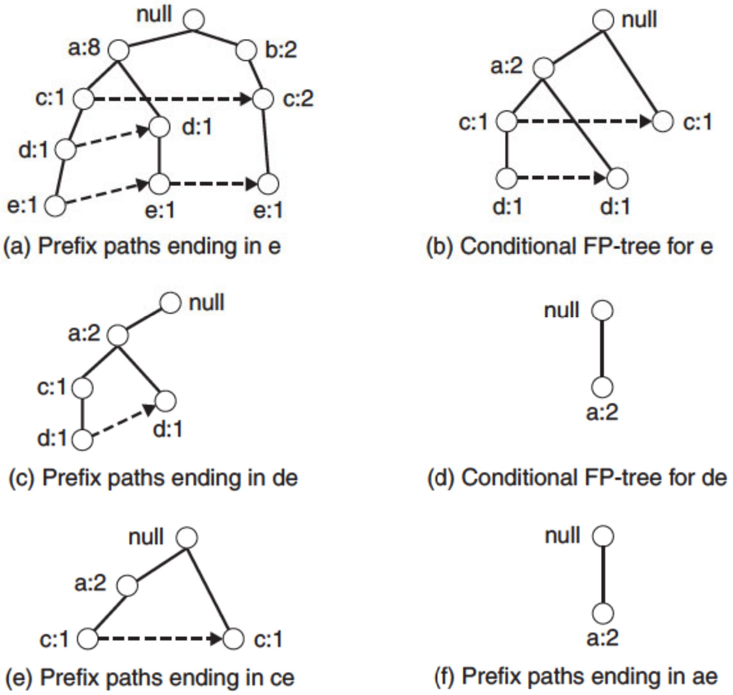
Figure 6.26. Decomposing the frequent itemset generation problem into multiple subproblems, where each subproblem involves finding frequent itemsets ending in e, d, c, b, and a.

Table 6.6. The list of frequent itemsets ordered by their corresponding suffixes.

Suffix	Frequent Itemsets
e	{e}, {d,e}, {a,d,e}, {c,e},{a,e}
d	{d}, {c,d}, {b,c,d}, {a,c,d}, {b,d}, {a,b,d}, {a,d}
c	{c}, {b,c}, {a,b,c}, {a,c}
b	{b}, {a,b}
a	{a}

After finding the frequent itemsets ending in e, the algorithm proceeds to look for frequent itemsets ending in d by processing the paths associated with node d. The corresponding paths are shown in Figure 6.26(b). This process continues until all the paths associated with nodes c, b, and finally a, are processed. The paths for these items are shown in Figures 6.26(c), (d), and (e), while their corresponding frequent itemsets are summarized in Table 6.6.

FP-growth finds all the frequent itemsets ending with a particular suffix by employing a divide-and-conquer strategy to split the problem into smaller subproblems. For example, suppose we are interested in finding all frequent



(a) Prefix paths ending in e

(b) Conditional FP-tree for e

(c) Prefix paths ending in de

(d) Conditional FP-tree for de

(e) Prefix paths ending in ce

(f) Prefix paths ending in ae

Figure 6.27. Example of applying the FP-growth algorithm to find frequent itemsets ending in e.

itemsets ending in e. To do this, we must first check whether the itemset $\{e\}$ itself is frequent. If it is frequent, we consider the subproblem of finding frequent itemsets ending in de, followed by ce, be, and ae. In turn, each of these subproblems are further decomposed into smaller subproblems. By merging the solutions obtained from the subproblems, all the frequent itemsets ending in e can be found. This divide-and-conquer approach is the key strategy employed by the FP-growth algorithm.

For a more concrete example on how to solve the subproblems, consider the task of finding frequent itemsets ending with e.

1. The first step is to gather all the paths containing node e. These initial paths are called **prefix paths** and are shown in Figure 6.27(a).

2. From the prefix paths shown in Figure 6.27(a), the support count for e is obtained by adding the support counts associated with node e. Assuming that the minimum support count is 2, $\{e\}$ is declared a frequent itemset because its support count is 3.

3. Because $\{e\}$ is frequent, the algorithm has to solve the subproblems of finding frequent itemsets ending in *de*, *ce*, *be*, and *ae*. Before solving these subproblems, it must first convert the prefix paths into a **conditional FP-tree**, which is structurally similar to an FP-tree, except it is used to find frequent itemsets ending with a particular suffix. A conditional FP-tree is obtained in the following way:

   (a) First, the support counts along the prefix paths must be updated because some of the counts include transactions that do not contain item *e*. For example, the rightmost path shown in Figure 6.27(a), `null` $\longrightarrow$ `b:2` $\longrightarrow$ `c:2` $\longrightarrow$ `e:1`, includes a transaction $\{b, c\}$ that does not contain item *e*. The counts along the prefix path must therefore be adjusted to 1 to reflect the actual number of transactions containing $\{b, c, e\}$.

   (b) The prefix paths are truncated by removing the nodes for *e*. These nodes can be removed because the support counts along the prefix paths have been updated to reflect only transactions that contain *e* and the subproblems of finding frequent itemsets ending in *de*, *ce*, *be*, and *ae* no longer need information about node *e*.

   (c) After updating the support counts along the prefix paths, some of the items may no longer be frequent. For example, the node *b* appears only once and has a support count equal to 1, which means that there is only one transaction that contains both *b* and *e*. Item *b* can be safely ignored from subsequent analysis because all itemsets ending in *be* must be infrequent.

   The conditional FP-tree for *e* is shown in Figure 6.27(b). The tree looks different than the original prefix paths because the frequency counts have been updated and the nodes *b* and *e* have been eliminated.

4. FP-growth uses the conditional FP-tree for *e* to solve the subproblems of finding frequent itemsets ending in *de*, *ce*, and *ae*. To find the frequent itemsets ending in *de*, the prefix paths for *d* are gathered from the conditional FP-tree for *e* (Figure 6.27(c)). By adding the frequency counts associated with node *d*, we obtain the support count for $\{d, e\}$. Since the support count is equal to 2, $\{d, e\}$ is declared a frequent itemset. Next, the algorithm constructs the conditional FP-tree for *de* using the approach described in step 3. After updating the support counts and removing the infrequent item *c*, the conditional FP-tree for *de* is shown in Figure 6.27(d). Since the conditional FP-tree contains only one item,

a, whose support is equal to *minsup*, the algorithm extracts the frequent itemset $\{a, d, e\}$ and moves on to the next subproblem, which is to generate frequent itemsets ending in ce. After processing the prefix paths for c, only $\{c, e\}$ is found to be frequent. The algorithm proceeds to solve the next subprogram and found $\{a, e\}$ to be the only frequent itemset remaining.

This example illustrates the divide-and-conquer approach used in the FP-growth algorithm. At each recursive step, a conditional FP-tree is constructed by updating the frequency counts along the prefix paths and removing all infrequent items. Because the subproblems are disjoint, FP-growth will not generate any duplicate itemsets. In addition, the counts associated with the nodes allow the algorithm to perform support counting while generating the common suffix itemsets.

FP-growth is an interesting algorithm because it illustrates how a compact representation of the transaction data set helps to efficiently generate frequent itemsets. In addition, for certain transaction data sets, FP-growth outperforms the standard *Apriori* algorithm by several orders of magnitude. The run-time performance of FP-growth depends on the **compaction factor** of the data set. If the resulting conditional FP-trees are very bushy (in the worst case, a full prefix tree), then the performance of the algorithm degrades significantly because it has to generate a large number of subproblems and merge the results returned by each subproblem.

6.7 Evaluation of Association Patterns

Association analysis algorithms have the potential to generate a large number of patterns. For example, although the data set shown in Table 6.1 contains only six items, it can produce up to hundreds of association rules at certain support and confidence thresholds. As the size and dimensionality of real commercial databases can be very large, we could easily end up with thousands or even millions of patterns, many of which might not be interesting. Sifting through the patterns to identify the most interesting ones is not a trivial task because "one person's trash might be another person's treasure." It is therefore important to establish a set of well-accepted criteria for evaluating the quality of association patterns.

The first set of criteria can be established through statistical arguments. Patterns that involve a set of mutually independent items or cover very few transactions are considered uninteresting because they may capture spurious relationships in the data. Such patterns can be eliminated by applying an

objective interestingness measure that uses statistics derived from data to determine whether a pattern is interesting. Examples of objective interestingness measures include support, confidence, and correlation.

The second set of criteria can be established through subjective arguments. A pattern is considered subjectively uninteresting unless it reveals unexpected information about the data or provides useful knowledge that can lead to profitable actions. For example, the rule $\{Butter\} \longrightarrow \{Bread\}$ may not be interesting, despite having high support and confidence values, because the relationship represented by the rule may seem rather obvious. On the other hand, the rule $\{Diapers\} \longrightarrow \{Beer\}$ is interesting because the relationship is quite unexpected and may suggest a new cross-selling opportunity for retailers. Incorporating subjective knowledge into pattern evaluation is a difficult task because it requires a considerable amount of prior information from the domain experts.

The following are some of the approaches for incorporating subjective knowledge into the pattern discovery task.

Visualization This approach requires a user-friendly environment to keep the human user in the loop. It also allows the domain experts to interact with the data mining system by interpreting and verifying the discovered patterns.

Template-based approach This approach allows the users to constrain the type of patterns extracted by the mining algorithm. Instead of reporting all the extracted rules, only rules that satisfy a user-specified template are returned to the users.

Subjective interestingness measure A subjective measure can be defined based on domain information such as concept hierarchy (to be discussed in Section 7.3) or profit margin of items. The measure can then be used to filter patterns that are obvious and non-actionable.

Readers interested in subjective interestingness measures may refer to resources listed in the bibliography at the end of this chapter.

6.7.1 Objective Measures of Interestingness

An objective measure is a data-driven approach for evaluating the quality of association patterns. It is domain-independent and requires minimal input from the users, other than to specify a threshold for filtering low-quality patterns. An objective measure is usually computed based on the frequency

Table 6.7. A 2-way contingency table for variables A and B.

	B	$\overline{B}$	
A	f_{11}	f_{10}	f_{1+}
$\overline{A}$	f_{01}	f_{00}	f_{0+}
	f_{+1}	f_{+0}	N

counts tabulated in a **contingency table**. Table 6.7 shows an example of a contingency table for a pair of binary variables, A and B. We use the notation $\overline{A}$ ($\overline{B}$) to indicate that A (B) is absent from a transaction. Each entry f_{ij} in this 2×2 table denotes a frequency count. For example, f_{11} is the number of times A and B appear together in the same transaction, while f_{01} is the number of transactions that contain B but not A. The row sum f_{1+} represents the support count for A, while the column sum f_{+1} represents the support count for B. Finally, even though our discussion focuses mainly on asymmetric binary variables, note that contingency tables are also applicable to other attribute types such as symmetric binary, nominal, and ordinal variables.

Limitations of the Support-Confidence Framework Existing association rule mining formulation relies on the support and confidence measures to eliminate uninteresting patterns. The drawback of support was previously described in Section 6.8, in which many potentially interesting patterns involving low support items might be eliminated by the support threshold. The drawback of confidence is more subtle and is best demonstrated with the following example.

Example 6.3. Suppose we are interested in analyzing the relationship between people who drink tea and coffee. We may gather information about the beverage preferences among a group of people and summarize their responses into a table such as the one shown in Table 6.8.

Table 6.8. Beverage preferences among a group of 1000 people.

	$Coffee$	$\overline{Coffee}$	
Tea	150	50	200
$\overline{Tea}$	650	150	800
	800	200	1000

The information given in this table can be used to evaluate the association rule $\{Tea\} \longrightarrow \{Coffee\}$. At first glance, it may appear that people who drink tea also tend to drink coffee because the rule's support (15%) and confidence (75%) values are reasonably high. This argument would have been acceptable except that the fraction of people who drink coffee, regardless of whether they drink tea, is 80%, while the fraction of tea drinkers who drink coffee is only 75%. Thus knowing that a person is a tea drinker actually decreases her probability of being a coffee drinker from 80% to 75%! The rule $\{Tea\} \longrightarrow \{Coffee\}$ is therefore misleading despite its high confidence value. ■

The pitfall of confidence can be traced to the fact that the measure ignores the support of the itemset in the rule consequent. Indeed, if the support of coffee drinkers is taken into account, we would not be surprised to find that many of the people who drink tea also drink coffee. What is more surprising is that the fraction of tea drinkers who drink coffee is actually less than the overall fraction of people who drink coffee, which points to an inverse relationship between tea drinkers and coffee drinkers.

Because of the limitations in the support-confidence framework, various objective measures have been used to evaluate the quality of association patterns. Below, we provide a brief description of these measures and explain some of their strengths and limitations.

Interest Factor The tea-coffee example shows that high-confidence rules can sometimes be misleading because the confidence measure ignores the support of the itemset appearing in the rule consequent. One way to address this problem is by applying a metric known as **lift**:

$$Lift = \frac{c(A \longrightarrow B)}{s(B)}, \tag{6.4}$$

which computes the ratio between the rule's confidence and the support of the itemset in the rule consequent. For binary variables, lift is equivalent to another objective measure called **interest factor**, which is defined as follows:

$$I(A, B) = \frac{s(A, B)}{s(A) \times s(B)} = \frac{Nf_{11}}{f_{1+}f_{+1}}. \tag{6.5}$$

Interest factor compares the frequency of a pattern against a baseline frequency computed under the statistical independence assumption. The baseline frequency for a pair of mutually independent variables is

$$\frac{f_{11}}{N} = \frac{f_{1+}}{N} \times \frac{f_{+1}}{N}, \quad \text{or equivalently,} \quad f_{11} = \frac{f_{1+}f_{+1}}{N}. \tag{6.6}$$

Table 6.9. Contingency tables for the word pairs ($\{p,q\}$ and $\{r,s\}$.

	p	$\overline{p}$	
q	880	50	930
$\overline{q}$	50	20	70
	930	70	1000

	r	$\overline{r}$	
s	20	50	70
$\overline{s}$	50	880	930
	70	930	1000

This equation follows from the standard approach of using simple fractions as estimates for probabilities. The fraction f_{11}/N is an estimate for the joint probability $P(A, B)$, while f_{1+}/N and f_{+1}/N are the estimates for $P(A)$ and $P(B)$, respectively. If A and B are statistically independent, then $P(A, B) = P(A) \times P(B)$, thus leading to the formula shown in Equation 6.6. Using Equations 6.5 and 6.6, we can interpret the measure as follows:

$$I(A,B) \begin{cases} = 1, & \text{if } A \text{ and } B \text{ are independent;} \\ > 1, & \text{if } A \text{ and } B \text{ are positively correlated;} \\ < 1, & \text{if } A \text{ and } B \text{ are negatively correlated.} \end{cases} \tag{6.7}$$

For the tea-coffee example shown in Table 6.8, $I = \frac{0.15}{0.2\times0.8} = 0.9375$, thus suggesting a slight negative correlation between tea drinkers and coffee drinkers.

Limitations of Interest Factor We illustrate the limitation of interest factor with an example from the text mining domain. In the text domain, it is reasonable to assume that the association between a pair of words depends on the number of documents that contain both words. For example, because of their stronger association, we expect the words **`data`** and **`mining`** to appear together more frequently than the words **`compiler`** and **`mining`** in a collection of computer science articles.

Table 6.9 shows the frequency of occurrences between two pairs of words, $\{p, q\}$ and $\{r, s\}$. Using the formula given in Equation 6.5, the interest factor for $\{p, q\}$ is 1.02 and for $\{r, s\}$ is 4.08. These results are somewhat troubling for the following reasons. Although p and q appear together in 88% of the documents, their interest factor is close to 1, which is the value when p and q are statistically independent. On the other hand, the interest factor for $\{r, s\}$ is higher than $\{p, q\}$ even though r and s seldom appear together in the same document. Confidence is perhaps the better choice in this situation because it considers the association between p and q (94.6%) to be much stronger than that between r and s (28.6%).

Correlation Analysis Correlation analysis is a statistical-based technique for analyzing relationships between a pair of variables. For continuous variables, correlation is defined using Pearson's correlation coefficient (see Equation 2.10 on page 77). For binary variables, correlation can be measured using the ϕ-coefficient, which is defined as

$$\phi = \frac{f_{11}f_{00} - f_{01}f_{10}}{\sqrt{f_{1+}f_{+1}f_{0+}f_{+0}}}. \tag{6.8}$$

The value of correlation ranges from -1 (perfect negative correlation) to $+1$ (perfect positive correlation). If the variables are statistically independent, then $\phi = 0$. For example, the correlation between the tea and coffee drinkers given in Table 6.8 is -0.0625.

Limitations of Correlation Analysis The drawback of using correlation can be seen from the word association example given in Table 6.9. Although the words p and q appear together more often than r and s, their ϕ-coefficients are identical, i.e., $\phi(p,q) = \phi(r,s) = 0.232$. This is because the ϕ-coefficient gives equal importance to both co-presence and co-absence of items in a transaction. It is therefore more suitable for analyzing symmetric binary variables. Another limitation of this measure is that it does not remain invariant when there are proportional changes to the sample size. This issue will be discussed in greater detail when we describe the properties of objective measures on page 377.

IS Measure IS is an alternative measure that has been proposed for handling asymmetric binary variables. The measure is defined as follows:

$$IS(A,B) = \sqrt{I(A,B) \times s(A,B)} = \frac{s(A,B)}{\sqrt{s(A)s(B)}}. \tag{6.9}$$

Note that IS is large when the interest factor and support of the pattern are large. For example, the value of IS for the word pairs $\{p,q\}$ and $\{r,s\}$ shown in Table 6.9 are 0.946 and 0.286, respectively. Contrary to the results given by interest factor and the ϕ-coefficient, the IS measure suggests that the association between $\{p,q\}$ is stronger than $\{r,s\}$, which agrees with what we expect from word associations in documents.

It is possible to show that IS is mathematically equivalent to the cosine measure for binary variables (see Equation 2.7 on page 75). In this regard, we

Table 6.10. Example of a contingency table for items p and q.

	q	$\overline{q}$	
p	800	100	900
$\overline{p}$	100	0	100
	900	100	1000

consider **A** and **B** as a pair of bit vectors, $\mathbf{A} \bullet \mathbf{B} = s(A, B)$ the dot product between the vectors, and $|\mathbf{A}| = \sqrt{s(A)}$ the magnitude of vector **A**. Therefore:

$$IS(A, B) = \frac{s(A, B)}{\sqrt{s(A) \times s(B)}} = \frac{\mathbf{A} \bullet \mathbf{B}}{|\mathbf{A}| \times |\mathbf{B}|} = cosine(\mathbf{A}, \mathbf{B}). \qquad (6.10)$$

The IS measure can also be expressed as the geometric mean between the confidence of association rules extracted from a pair of binary variables:

$$IS(A, B) = \sqrt{\frac{s(A, B)}{s(A)} \times \frac{s(A, B)}{s(B)}} = \sqrt{c(A \rightarrow B) \times c(B \rightarrow A)}. \qquad (6.11)$$

Because the geometric mean between any two numbers is always closer to the smaller number, the IS value of an itemset $\{p, q\}$ is low whenever one of its rules, $p \longrightarrow q$ or $q \longrightarrow p$, has low confidence.

Limitations of IS Measure The IS value for a pair of independent itemsets, A and B, is

$$IS_{\text{indep}}(A, B) = \frac{s(A, B)}{\sqrt{s(A) \times s(B)}} = \frac{s(A) \times s(B)}{\sqrt{s(A) \times s(B)}} = \sqrt{s(A) \times s(B)}.$$

Since the value depends on $s(A)$ and $s(B)$, IS shares a similar problem as the confidence measure—that the value of the measure can be quite large, even for uncorrelated and negatively correlated patterns. For example, despite the large IS value between items p and q given in Table 6.10 (0.889), it is still less than the expected value when the items are statistically independent ($IS_{\text{indep}} = 0.9$).

Alternative Objective Interestingness Measures

Besides the measures we have described so far, there are other alternative measures proposed for analyzing relationships between pairs of binary variables. These measures can be divided into two categories, **symmetric** and **asymmetric** measures. A measure M is symmetric if $M(A \longrightarrow B) = M(B \longrightarrow A)$. For example, interest factor is a symmetric measure because its value is identical for the rules $A \longrightarrow B$ and $B \longrightarrow A$. In contrast, confidence is an asymmetric measure since the confidence for $A \longrightarrow B$ and $B \longrightarrow A$ may not be the same. Symmetric measures are generally used for evaluating itemsets, while asymmetric measures are more suitable for analyzing association rules. Tables 6.11 and 6.12 provide the definitions for some of these measures in terms of the frequency counts of a 2×2 contingency table.

Consistency among Objective Measures

Given the wide variety of measures available, it is reasonable to question whether the measures can produce similar ordering results when applied to a set of association patterns. If the measures are consistent, then we can choose any one of them as our evaluation metric. Otherwise, it is important to understand what their differences are in order to determine which measure is more suitable for analyzing certain types of patterns.

Table 6.11. Examples of symmetric objective measures for the itemset $\{A, B\}$.

Measure (Symbol)	Definition
Correlation (ϕ)	$\frac{Nf_{11} - f_{1+}f_{+1}}{\sqrt{f_{1+}f_{+1}f_{0+}f_{+0}}}$
Odds ratio (α)	$(f_{11}f_{00})/(f_{10}f_{01})$
Kappa (κ)	$\frac{Nf_{11} + Nf_{00} - f_{1+}f_{+1} - f_{0+}f_{+0}}{N^2 - f_{1+}f_{+1} - f_{0+}f_{+0}}$
Interest (I)	$(Nf_{11})/(f_{1+}f_{+1})$
Cosine (IS)	$(f_{11})/(\sqrt{f_{1+}f_{+1}})$
Piatetsky-Shapiro (PS)	$\frac{f_{11}}{N} - \frac{f_{1+}f_{+1}}{N^2}$
Collective strength (S)	$\frac{f_{11} + f_{00}}{f_{1+}f_{+1} + f_{0+}f_{+0}} \times \frac{N - f_{1+}f_{+1} - f_{0+}f_{+0}}{N - f_{11} - f_{00}}$
Jaccard (ζ)	$f_{11}/(f_{1+} + f_{+1} - f_{11})$
All-confidence (h)	$\min\left[\frac{f_{11}}{f_{1+}}, \frac{f_{11}}{f_{+1}}\right]$

Table 6.12. Examples of asymmetric objective measures for the rule $A \longrightarrow B$.

Measure (Symbol)	Definition
Goodman-Kruskal (λ)	$(\sum_j \max_k f_{jk} - max_k f_{+k})/(N - \max_k f_{+k})$
Mutual Information (M)	$(\sum_i \sum_j \frac{f_{ij}}{N} \log \frac{N f_{ij}}{f_{i+} f_{+j}})/(-\sum_i \frac{f_{i+}}{N} \log \frac{f_{i+}}{N})$
J-Measure (J)	$\frac{f_{11}}{N} \log \frac{N f_{11}}{f_{1+} f_{+1}} + \frac{f_{10}}{N} \log \frac{N f_{10}}{f_{1+} f_{+0}}$
Gini index (G)	$\frac{f_{1+}}{N} \times (\frac{f_{11}}{f_{1+}})^2 + (\frac{f_{10}}{f_{1+}})^2] - (\frac{f_{+1}}{N})^2$
	$+ \frac{f_{0+}}{N} \times [(\frac{f_{01}}{f_{0+}})^2 + (\frac{f_{00}}{f_{0+}})^2] - (\frac{f_{+0}}{N})^2$
Laplace (L)	$(f_{11} + 1)/(f_{1+} + 2)$
Conviction (V)	$(f_{1+} f_{+0})/(N f_{10})$
Certainty factor (F)	$(\frac{f_{11}}{f_{1+}} - \frac{f_{+1}}{N})/(1 - \frac{f_{+1}}{N})$
Added Value (AV)	$\frac{f_{11}}{f_{1+}} - \frac{f_{+1}}{N}$

Table 6.13. Example of contingency tables.

Example	f_{11}	f_{10}	f_{01}	f_{00}
E_1	8123	83	424	1370
E_2	8330	2	622	1046
E_3	3954	3080	5	2961
E_4	2886	1363	1320	4431
E_5	1500	2000	500	6000
E_6	4000	2000	1000	3000
E_7	9481	298	127	94
E_8	4000	2000	2000	2000
E_9	7450	2483	4	63
E_{10}	61	2483	4	7452

Suppose the symmetric and asymmetric measures are applied to rank the ten contingency tables shown in Table 6.13. These contingency tables are chosen to illustrate the differences among the existing measures. The ordering produced by these measures are shown in Tables 6.14 and 6.15, respectively (with 1 as the most interesting and 10 as the least interesting table). Although some of the measures appear to be consistent with each other, there are certain measures that produce quite different ordering results. For example, the rankings given by the ϕ-coefficient agree with those provided by κ and collective strength, but are somewhat different than the rankings produced by interest

Table 6.14. Rankings of contingency tables using the symmetric measures given in Table 6.11.

	ϕ	α	κ	I	IS	PS	S	ζ	h
E_1	1	3	1	6	2	2	1	2	2
E_2	2	1	2	7	3	5	2	3	3
E_3	3	2	4	4	5	1	3	6	8
E_4	4	8	3	3	7	3	4	7	5
E_5	5	7	6	2	9	6	6	9	9
E_6	6	9	5	5	6	4	5	5	7
E_7	7	6	7	9	1	8	7	1	1
E_8	8	10	8	8	8	7	8	8	7
E_9	9	4	9	10	4	9	9	4	4
E_{10}	10	5	10	1	10	10	10	10	10

Table 6.15. Rankings of contingency tables using the asymmetric measures given in Table 6.12.

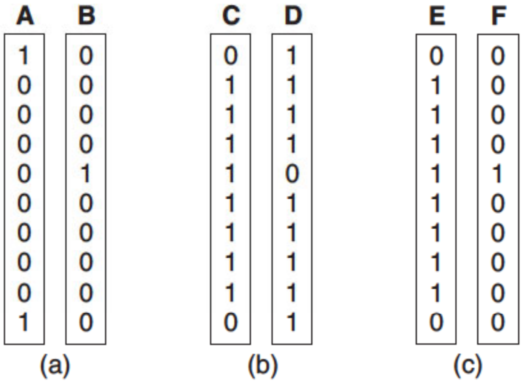
	λ	M	J	G	L	V	F	AV
E_1	1	1	1	1	4	2	2	5
E_2	2	2	2	3	5	1	1	6
E_3	5	3	5	2	2	6	6	4
E_4	4	6	3	4	9	3	3	1
E_5	9	7	4	6	8	5	5	2
E_6	3	8	6	5	7	4	4	3
E_7	7	5	9	8	3	7	7	9
E_8	8	9	7	7	10	8	8	7
E_9	6	4	10	9	1	9	9	10
E_{10}	10	10	8	10	6	10	10	8

factor and odds ratio. Furthermore, a contingency table such as E_{10} is ranked lowest according to the ϕ-coefficient, but highest according to interest factor.

Properties of Objective Measures

The results shown in Table 6.14 suggest that a significant number of the measures provide conflicting information about the quality of a pattern. To understand their differences, we need to examine the properties of these measures.

Inversion Property Consider the bit vectors shown in Figure 6.28. The 0/1 bit in each column vector indicates whether a transaction (row) contains a particular item (column). For example, the vector **A** indicates that item a



A	B	C	D	E	F
1	0	0	1	0	0
0	0	1	1	1	0
0	0	1	1	1	0
0	0	1	1	1	0
0	1	1	0	1	1
0	0	1	1	1	0
0	0	1	1	1	0
0	0	1	1	1	0
0	0	1	1	1	0
1	0	0	1	0	0

Figure 6.28. Effect of the inversion operation. The vectors C and E are inversions of vector A, while the vector D is an inversion of vectors B and F.

belongs to the first and last transactions, whereas the vector **B** indicates that item b is contained only in the fifth transaction. The vectors **C** and **E** are in fact related to the vector **A**—their bits have been inverted from 0's (absence) to 1's (presence), and vice versa. Similarly, **D** is related to vectors **B** and **F** by inverting their bits. The process of flipping a bit vector is called **inversion**. If a measure is invariant under the inversion operation, then its value for the vector pair $(\mathbf{C}, \mathbf{D})$ should be identical to its value for $(\mathbf{A}, \mathbf{B})$. The inversion property of a measure can be tested as follows.

Definition 6.6 (Inversion Property). An objective measure M is invariant under the inversion operation if its value remains the same when exchanging the frequency counts f_{11} with f_{00} and f_{10} with f_{01}.

Among the measures that remain invariant under this operation include the ϕ-coefficient, odds ratio, κ, and collective strength. These measures may not be suitable for analyzing asymmetric binary data. For example, the ϕ-coefficient between **C** and **D** is identical to the ϕ-coefficient between **A** and **B**, even though items c and d appear together more frequently than a and b. Furthermore, the ϕ-coefficient between **C** and **D** is less than that between **E** and **F** even though items e and f appear together only once! We had previously raised this issue when discussing the limitations of the ϕ-coefficient on page 375. For asymmetric binary data, measures that do not remain invariant under the inversion operation are preferred. Some of the non-invariant measures include interest factor, IS, PS, and the Jaccard coefficient.

Null Addition Property Suppose we are interested in analyzing the relationship between a pair of words, such as `data` and `mining`, in a set of documents. If a collection of articles about ice fishing is added to the data set, should the association between `data` and `mining` be affected? This process of adding unrelated data (in this case, documents) to a given data set is known as the **null addition** operation.

Definition 6.7 (Null Addition Property). An objective measure M is invariant under the null addition operation if it is not affected by increasing f_{00}, while all other frequencies in the contingency table stay the same.

For applications such as document analysis or market basket analysis, the measure is expected to remain invariant under the null addition operation. Otherwise, the relationship between words may disappear simply by adding enough documents that do not contain both words! Examples of measures that satisfy this property include cosine (IS) and Jaccard (ξ) measures, while those that violate this property include interest factor, PS, odds ratio, and the ϕ-coefficient.

Scaling Property Table 6.16 shows the contingency tables for gender and the grades achieved by students enrolled in a particular course in 1993 and 2004. The data in these tables showed that the number of male students has doubled since 1993, while the number of female students has increased by a factor of 3. However, the male students in 2004 are not performing any better than those in 1993 because the ratio of male students who achieve a high grade to those who achieve a low grade is still the same, i.e., 3:4. Similarly, the female students in 2004 are performing no better than those in 1993. The association between grade and gender is expected to remain unchanged despite changes in the sampling distribution.

Table 6.16. The grade-gender example.

	Male	Female	
High	30	20	50
Low	40	10	50
	70	30	100

(a) Sample data from 1993.

	Male	Female	
High	60	60	120
Low	80	30	110
	140	90	230

(b) Sample data from 2004.

Table 6.17. Properties of symmetric measures.

Symbol	Measure	Inversion	Null Addition	Scaling
ϕ	ϕ-coefficient	Yes	No	No
α	odds ratio	Yes	No	Yes
κ	Cohen's	Yes	No	No
I	Interest	No	No	No
IS	Cosine	No	Yes	No
PS	Piatetsky-Shapiro's	Yes	No	No
S	Collective strength	Yes	No	No
ζ	Jaccard	No	Yes	No
h	All-confidence	No	No	No
s	Support	No	No	No

Definition 6.8 (Scaling Invariance Property). An objective measure M is invariant under the row/column scaling operation if $M(T) = M(T')$, where T is a contingency table with frequency counts $[f_{11};\ f_{10};\ f_{01};\ f_{00}]$, T' is a contingency table with scaled frequency counts $[k_1k_3f_{11};\ k_2k_3f_{10};\ k_1k_4f_{01};\ k_2k_4f_{00}]$, and k_1, k_2, k_3, k_4 are positive constants.

From Table 6.17, notice that only the odds ratio (α) is invariant under the row and column scaling operations. All other measures such as the ϕ-coefficient, κ, IS, interest factor, and collective strength (S) change their values when the rows and columns of the contingency table are rescaled. Although we do not discuss the properties of asymmetric measures (such as confidence, J-measure, Gini index, and conviction), it is clear that such measures do not preserve their values under inversion and row/column scaling operations, but are invariant under the null addition operation.

6.7.2 Measures beyond Pairs of Binary Variables

The measures shown in Tables 6.11 and 6.12 are defined for pairs of binary variables (e.g., 2-itemsets or association rules). However, many of them, such as support and all-confidence, are also applicable to larger-sized itemsets. Other measures, such as interest factor, IS, PS, and Jaccard coefficient, can be extended to more than two variables using the frequency tables tabulated in a multidimensional contingency table. An example of a three-dimensional contingency table for a, b, and c is shown in Table 6.18. Each entry f_{ijk} in this table represents the number of transactions that contain a particular combination of items a, b, and c. For example, f_{101} is the number of transactions that contain a and c, but not b. On the other hand, a marginal frequency

Table 6.18. Example of a three-dimensional contingency table.

c	b	$\overline{b}$	
a	f_{111}	f_{101}	f_{1+1}
$\overline{a}$	f_{011}	f_{001}	f_{0+1}
	f_{+11}	f_{+01}	f_{++1}

$\overline{c}$	b	$\overline{b}$	
a	f_{110}	f_{100}	f_{1+0}
$\overline{a}$	f_{010}	f_{000}	f_{0+0}
	f_{+10}	f_{+00}	f_{++0}

such as f_{1+1} is the number of transactions that contain a and c, irrespective of whether b is present in the transaction.

Given a k-itemset $\{i_1, i_2, \ldots, i_k\}$, the condition for statistical independence can be stated as follows:

$$f_{i_1 i_2 \ldots i_k} = \frac{f_{i_1 + \ldots +} \times f_{+ i_2 \ldots +} \times \ldots \times f_{++ \ldots i_k}}{N^{k-1}}. \tag{6.12}$$

With this definition, we can extend objective measures such as interest factor and PS, which are based on deviations from statistical independence, to more than two variables:

$$\begin{aligned} I &= \frac{N^{k-1} \times f_{i_1 i_2 \ldots i_k}}{f_{i_1 + \ldots +} \times f_{+ i_2 \ldots +} \times \ldots \times f_{++ \ldots i_k}} \\ PS &= \frac{f_{i_1 i_2 \ldots i_k}}{N} - \frac{f_{i_1 + \ldots +} \times f_{+ i_2 \ldots +} \times \ldots \times f_{++ \ldots i_k}}{N^k} \end{aligned}$$

Another approach is to define the objective measure as the maximum, minimum, or average value for the associations between pairs of items in a pattern. For example, given a k-itemset $X = \{i_1, i_2, \ldots, i_k\}$, we may define the ϕ-coefficient for X as the average ϕ-coefficient between every pair of items (i_p, i_q) in X. However, because the measure considers only pairwise associations, it may not capture all the underlying relationships within a pattern.

Analysis of multidimensional contingency tables is more complicated because of the presence of partial associations in the data. For example, some associations may appear or disappear when conditioned upon the value of certain variables. This problem is known as **Simpson's paradox** and is described in the next section. More sophisticated statistical techniques are available to analyze such relationships, e.g., loglinear models, but these techniques are beyond the scope of this book.

Table 6.19. A two-way contingency table between the sale of high-definition television and exercise machine.

Buy	Buy Exercise Machine		
HDTV	Yes	No	
Yes	99	81	180
No	54	66	120
	153	147	300

Table 6.20. Example of a three-way contingency table.

Customer	Buy	Buy Exercise Machine		Total
Group	HDTV	Yes	No	
College Students	Yes	1	9	10
	No	4	30	34
Working Adult	Yes	98	72	170
	No	50	36	86

6.7.3 Simpson's Paradox

It is important to exercise caution when interpreting the association between variables because the observed relationship may be influenced by the presence of other confounding factors, i.e., hidden variables that are not included in the analysis. In some cases, the hidden variables may cause the observed relationship between a pair of variables to disappear or reverse its direction, a phenomenon that is known as Simpson's paradox. We illustrate the nature of this paradox with the following example.

Consider the relationship between the sale of high-definition television (HDTV) and exercise machine, as shown in Table 6.19. The rule $\{\text{HDTV=Yes}\} \longrightarrow \{\text{Exercise machine=Yes}\}$ has a confidence of $99/180 = 55\%$ and the rule $\{\text{HDTV=No}\} \longrightarrow \{\text{Exercise machine=Yes}\}$ has a confidence of $54/120 = 45\%$. Together, these rules suggest that customers who buy high-definition televisions are more likely to buy exercise machines than those who do not buy high-definition televisions.

However, a deeper analysis reveals that the sales of these items depend on whether the customer is a college student or a working adult. Table 6.20 summarizes the relationship between the sale of HDTVs and exercise machines among college students and working adults. Notice that the support counts given in the table for college students and working adults sum up to the frequencies shown in Table 6.19. Furthermore, there are more working adults than college students who buy these items. For college students:

$$\begin{aligned} c\big(\{\text{HDTV=Yes}\} \longrightarrow \{\text{Exercise machine=Yes}\}\big) &= 1/10 = 10\%, \\ c\big(\{\text{HDTV=No}\} \longrightarrow \{\text{Exercise machine=Yes}\}\big) &= 4/34 = 11.8\%, \end{aligned}$$

while for working adults:

$$\begin{aligned} c\big(\{\text{HDTV=Yes}\} \longrightarrow \{\text{Exercise machine=Yes}\}\big) &= 98/170 = 57.7\%, \\ c\big(\{\text{HDTV=No}\} \longrightarrow \{\text{Exercise machine=Yes}\}\big) &= 50/86 = 58.1\%. \end{aligned}$$

The rules suggest that, for each group, customers who do not buy high-definition televisions are more likely to buy exercise machines, which contradict the previous conclusion when data from the two customer groups are pooled together. Even if alternative measures such as correlation, odds ratio, or interest are applied, we still find that the sale of HDTV and exercise machine is positively correlated in the combined data but is negatively correlated in the stratified data (see Exercise 20 on page 414). The reversal in the direction of association is known as Simpson's paradox.

The paradox can be explained in the following way. Notice that most customers who buy HDTVs are working adults. Working adults are also the largest group of customers who buy exercise machines. Because nearly 85% of the customers are working adults, the observed relationship between HDTV and exercise machine turns out to be stronger in the combined data than what it would have been if the data is stratified. This can also be illustrated mathematically as follows. Suppose

$$a/b < c/d \text{ and } p/q < r/s,$$

where a/b and p/q may represent the confidence of the rule $A \longrightarrow B$ in two different strata, while c/d and r/s may represent the confidence of the rule $\overline{A} \longrightarrow B$ in the two strata. When the data is pooled together, the confidence values of the rules in the combined data are $(a+p)/(b+q)$ and $(c+r)/(d+s)$, respectively. Simpson's paradox occurs when

$$\frac{a+p}{b+q} > \frac{c+r}{d+s},$$

thus leading to the wrong conclusion about the relationship between the variables. The lesson here is that proper stratification is needed to avoid generating spurious patterns resulting from Simpson's paradox. For example, market

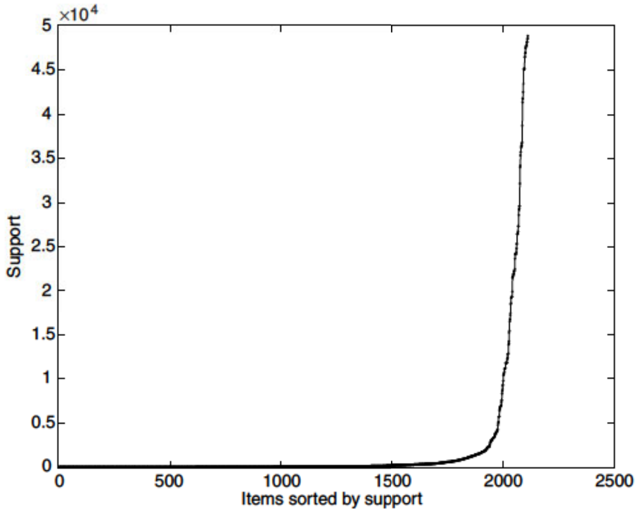


Figure 6.29. Support distribution of items in the census data set.

basket data from a major supermarket chain should be stratified according to store locations, while medical records from various patients should be stratified according to confounding factors such as age and gender.

6.8 Effect of Skewed Support Distribution

The performances of many association analysis algorithms are influenced by properties of their input data. For example, the computational complexity of the *Apriori* algorithm depends on properties such as the number of items in the data and average transaction width. This section examines another important property that has significant influence on the performance of association analysis algorithms as well as the quality of extracted patterns. More specifically, we focus on data sets with skewed support distributions, where most of the items have relatively low to moderate frequencies, but a small number of them have very high frequencies.

An example of a real data set that exhibits such a distribution is shown in Figure 6.29. The data, taken from the PUMS (Public Use Microdata Sample) census data, contains 49,046 records and 2113 asymmetric binary variables. We shall treat the asymmetric binary variables as items and records as transactions in the remainder of this section. While more than 80% of the items have support less than 1%, a handful of them have support greater than 90%.

Table 6.21. Grouping the items in the census data set based on their support values.

Group	G_1	G_2	G_3
Support	$< 1\%$	$1\% - 90\%$	$> 90\%$
Number of Items	1735	358	20

To illustrate the effect of skewed support distribution on frequent itemset mining, we divide the items into three groups, G_1, G_2, and G_3, according to their support levels. The number of items that belong to each group is shown in Table 6.21.

Choosing the right support threshold for mining this data set can be quite tricky. If we set the threshold too high (e.g., 20%), then we may miss many interesting patterns involving the low support items from G_1. In market basket analysis, such low support items may correspond to expensive products (such as jewelry) that are seldom bought by customers, but whose patterns are still interesting to retailers. Conversely, when the threshold is set too low, it becomes difficult to find the association patterns due to the following reasons. First, the computational and memory requirements of existing association analysis algorithms increase considerably with low support thresholds. Second, the number of extracted patterns also increases substantially with low support thresholds. Third, we may extract many spurious patterns that relate a high-frequency item such as milk to a low-frequency item such as caviar. Such patterns, which are called **cross-support** patterns, are likely to be spurious because their correlations tend to be weak. For example, at a support threshold equal to 0.05%, there are 18,847 frequent pairs involving items from G_1 and G_3. Out of these, 93% of them are cross-support patterns; i.e., the patterns contain items from both G_1 and G_3. The maximum correlation obtained from the cross-support patterns is 0.029, which is much lower than the maximum correlation obtained from frequent patterns involving items from the same group (which is as high as 1.0). Similar statement can be made about many other interestingness measures discussed in the previous section. This example shows that a large number of weakly correlated cross-support patterns can be generated when the support threshold is sufficiently low. Before presenting a methodology for eliminating such patterns, we formally define the concept of cross-support patterns.

Definition 6.9 (Cross-Support Pattern). A cross-support pattern is an itemset $X = \{i_1, i_2, \ldots, i_k\}$ whose support ratio

$$r(X) = \frac{\min\left[s(i_1), s(i_2), \ldots, s(i_k)\right]}{\max\left[s(i_1), s(i_2), \ldots, s(i_k)\right]}, \tag{6.13}$$

is less than a user-specified threshold h_c.

Example 6.4. Suppose the support for milk is 70%, while the support for sugar is 10% and caviar is 0.04%. Given $h_c = 0.01$, the frequent itemset {milk, sugar, caviar} is a cross-support pattern because its support ratio is

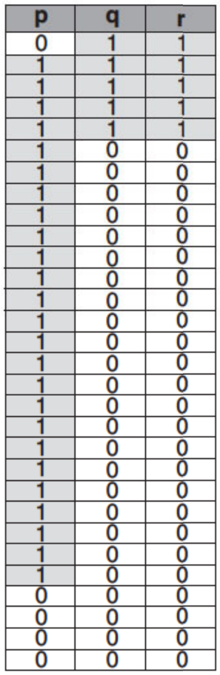
$$r = \frac{\min\left[0.7, 0.1, 0.0004\right]}{\max\left[0.7, 0.1, 0.0004\right]} = \frac{0.0004}{0.7} = 0.00058 < 0.01.$$

■

Existing measures such as support and confidence may not be sufficient to eliminate cross-support patterns, as illustrated by the data set shown in Figure 6.30. Assuming that $h_c = 0.3$, the itemsets $\{p, q\}$, $\{p, r\}$, and $\{p, q, r\}$ are cross-support patterns because their support ratios, which are equal to 0.2, are less than the threshold h_c. Although we can apply a high support threshold, say, 20%, to eliminate the cross-support patterns, this may come at the expense of discarding other interesting patterns such as the strongly correlated itemset, $\{q, r\}$ that has support equal to 16.7%.

Confidence pruning also does not help because the confidence of the rules extracted from cross-support patterns can be very high. For example, the confidence for $\{q\} \longrightarrow \{p\}$ is 80% even though $\{p, q\}$ is a cross-support pattern. The fact that the cross-support pattern can produce a high-confidence rule should not come as a surprise because one of its items (p) appears very frequently in the data. Therefore, p is expected to appear in many of the transactions that contain q. Meanwhile, the rule $\{q\} \longrightarrow \{r\}$ also has high confidence even though $\{q, r\}$ is not a cross-support pattern. This example demonstrates the difficulty of using the confidence measure to distinguish between rules extracted from cross-support and non-cross-support patterns.

Returning to the previous example, notice that the rule $\{p\} \longrightarrow \{q\}$ has very low confidence because most of the transactions that contain p do not contain q. In contrast, the rule $\{r\} \longrightarrow \{q\}$, which is derived from the pattern $\{q, r\}$, has very high confidence. This observation suggests that cross-support patterns can be detected by examining the lowest confidence rule that can be extracted from a given itemset. The proof of this statement can be understood as follows.



p	q	r
0	1	1
1	1	1
1	1	1
1	1	1
1	1	1
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Figure 6.30. A transaction data set containing three items, p, q, and r, where p is a high support item and q and r are low support items.

1. Recall the following anti-monotone property of confidence:

$$conf(\{i_1 i_2\} \longrightarrow \{i_3, i_4, \ldots, i_k\}) \leq conf(\{i_1 i_2 i_3\} \longrightarrow \{i_4, i_5, \ldots, i_k\}).$$

   This property suggests that confidence never increases as we shift more items from the left- to the right-hand side of an association rule. Because of this property, the lowest confidence rule extracted from a frequent itemset contains only one item on its left-hand side. We denote the set of all rules with only one item on its left-hand side as R_1.

2. Given a frequent itemset $\{i_1, i_2, \ldots, i_k\}$, the rule

$$\{i_j\} \longrightarrow \{i_1, i_2, \ldots, i_{j-1}, i_{j+1}, \ldots, i_k\}$$

   has the lowest confidence in R_1 if $s(i_j) = \max\big[s(i_1), s(i_2), \ldots, s(i_k)\big]$. This follows directly from the definition of confidence as the ratio between the rule's support and the support of the rule antecedent.

3. Summarizing the previous points, the lowest confidence attainable from a frequent itemset $\{i_1, i_2, \ldots, i_k\}$ is

$$\frac{s(\{i_1, i_2, \ldots, i_k\})}{\max\left[s(i_1), s(i_2), \ldots, s(i_k)\right]}.$$

   This expression is also known as the **h-confidence** or **all-confidence** measure. Because of the anti-monotone property of support, the numerator of the h-confidence measure is bounded by the minimum support of any item that appears in the frequent itemset. In other words, the h-confidence of an itemset $X = \{i_1, i_2, \ldots, i_k\}$ must not exceed the following expression:

$$\text{h-confidence}(X) \leq \frac{\min\left[s(i_1), s(i_2), \ldots, s(i_k)\right]}{\max\left[s(i_1), s(i_2), \ldots, s(i_k)\right]}.$$

   Note the equivalence between the upper bound of h-confidence and the support ratio (r) given in Equation 6.13. Because the support ratio for a cross-support pattern is always less than h_c, the h-confidence of the pattern is also guaranteed to be less than h_c.

Therefore, cross-support patterns can be eliminated by ensuring that the h-confidence values for the patterns exceed h_c. As a final note, it is worth mentioning that the advantages of using h-confidence go beyond eliminating cross-support patterns. The measure is also anti-monotone, i.e.,

$$\text{h-confidence}(\{i_1, i_2, \ldots, i_k\}) \geq \text{h-confidence}(\{i_1, i_2, \ldots, i_{k+1}\}),$$

and thus can be incorporated directly into the mining algorithm. Furthermore, h-confidence ensures that the items contained in an itemset are strongly associated with each other. For example, suppose the h-confidence of an itemset X is 80%. If one of the items in X is present in a transaction, there is at least an 80% chance that the rest of the items in X also belong to the same transaction. Such strongly associated patterns are called **hyperclique patterns**.

6.9 Bibliographic Notes

The association rule mining task was first introduced by Agrawal et al. in [228, 229] to discover interesting relationships among items in market basket

8

Cluster Analysis: Basic Concepts and Algorithms

Cluster analysis divides data into groups (clusters) that are meaningful, useful, or both. If meaningful groups are the goal, then the clusters should capture the natural structure of the data. In some cases, however, cluster analysis is only a useful starting point for other purposes, such as data summarization. Whether for understanding or utility, cluster analysis has long played an important role in a wide variety of fields: psychology and other social sciences, biology, statistics, pattern recognition, information retrieval, machine learning, and data mining.

There have been many applications of cluster analysis to practical problems. We provide some specific examples, organized by whether the purpose of the clustering is understanding or utility.

Clustering for Understanding Classes, or conceptually meaningful groups of objects that share common characteristics, play an important role in how people analyze and describe the world. Indeed, human beings are skilled at dividing objects into groups (clustering) and assigning particular objects to these groups (classification). For example, even relatively young children can quickly label the objects in a photograph as buildings, vehicles, people, animals, plants, etc. In the context of understanding data, clusters are potential classes and cluster analysis is the study of techniques for automatically finding classes. The following are some examples:

- **Biology.** Biologists have spent many years creating a taxonomy (hierarchical classification) of all living things: kingdom, phylum, class, order, family, genus, and species. Thus, it is perhaps not surprising that much of the early work in cluster analysis sought to create a discipline of mathematical taxonomy that could automatically find such classification structures. More recently, biologists have applied clustering to analyze the large amounts of genetic information that are now available. For example, clustering has been used to find groups of genes that have similar functions.

- **Information Retrieval.** The World Wide Web consists of billions of Web pages, and the results of a query to a search engine can return thousands of pages. Clustering can be used to group these search results into a small number of clusters, each of which captures a particular aspect of the query. For instance, a query of "movie" might return Web pages grouped into categories such as reviews, trailers, stars, and theaters. Each category (cluster) can be broken into subcategories (subclusters), producing a hierarchical structure that further assists a user's exploration of the query results.

- **Climate.** Understanding the Earth's climate requires finding patterns in the atmosphere and ocean. To that end, cluster analysis has been applied to find patterns in the atmospheric pressure of polar regions and areas of the ocean that have a significant impact on land climate.

- **Psychology and Medicine.** An illness or condition frequently has a number of variations, and cluster analysis can be used to identify these different subcategories. For example, clustering has been used to identify different types of depression. Cluster analysis can also be used to detect patterns in the spatial or temporal distribution of a disease.

- **Business.** Businesses collect large amounts of information on current and potential customers. Clustering can be used to segment customers into a small number of groups for additional analysis and marketing activities.

Clustering for Utility Cluster analysis provides an abstraction from individual data objects to the clusters in which those data objects reside. Additionally, some clustering techniques characterize each cluster in terms of a cluster prototype; i.e., a data object that is representative of the other objects in the cluster. These cluster prototypes can be used as the basis for a number of data analysis or data processing techniques. Therefore, in the context of utility, cluster analysis is the study of techniques for finding the most representative cluster prototypes.

- **Summarization.** Many data analysis techniques, such as regression or PCA, have a time or space complexity of $O(m^2)$ or higher (where m is the number of objects), and thus, are not practical for large data sets. However, instead of applying the algorithm to the entire data set, it can be applied to a reduced data set consisting only of cluster prototypes. Depending on the type of analysis, the number of prototypes, and the accuracy with which the prototypes represent the data, the results can be comparable to those that would have been obtained if all the data could have been used.

- **Compression.** Cluster prototypes can also be used for data compression. In particular, a table is created that consists of the prototypes for each cluster; i.e., each prototype is assigned an integer value that is its position (index) in the table. Each object is represented by the index of the prototype associated with its cluster. This type of compression is known as **vector quantization** and is often applied to image, sound, and video data, where (1) many of the data objects are highly similar to one another, (2) some loss of information is acceptable, and (3) a substantial reduction in the data size is desired.

- **Efficiently Finding Nearest Neighbors.** Finding nearest neighbors can require computing the pairwise distance between all points. Often clusters and their cluster prototypes can be found much more efficiently. If objects are relatively close to the prototype of their cluster, then we can use the prototypes to reduce the number of distance computations that are necessary to find the nearest neighbors of an object. Intuitively, if two cluster prototypes are far apart, then the objects in the corresponding clusters cannot be nearest neighbors of each other. Consequently, to find an object's nearest neighbors it is only necessary to compute the distance to objects in nearby clusters, where the nearness of two clusters is measured by the distance between their prototypes. This idea is made more precise in Exercise 25 on page 94.

This chapter provides an introduction to cluster analysis. We begin with a high-level overview of clustering, including a discussion of the various approaches to dividing objects into sets of clusters and the different types of clusters. We then describe three specific clustering techniques that represent broad categories of algorithms and illustrate a variety of concepts: K-means, agglomerative hierarchical clustering, and DBSCAN. The final section of this chapter is devoted to cluster validity—methods for evaluating the goodness of the clusters produced by a clustering algorithm. More advanced clustering concepts and algorithms will be discussed in Chapter 9. Whenever possible, we discuss the strengths and weaknesses of different schemes. In addition, the bibliographic notes provide references to relevant books and papers that explore cluster analysis in greater depth.

8.1 Overview

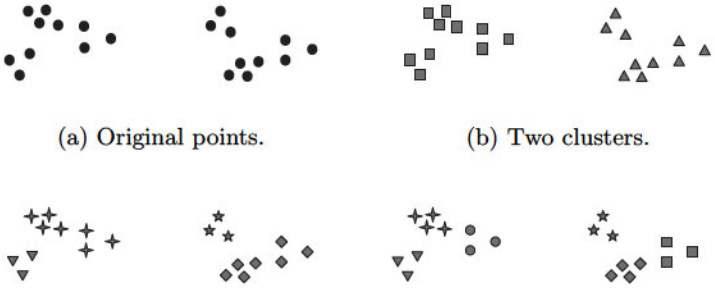
Before discussing specific clustering techniques, we provide some necessary background. First, we further define cluster analysis, illustrating why it is difficult and explaining its relationship to other techniques that group data. Then we explore two important topics: (1) different ways to group a set of objects into a set of clusters, and (2) types of clusters.

8.1.1 What Is Cluster Analysis?

Cluster analysis groups data objects based only on information found in the data that describes the objects and their relationships. The goal is that the objects within a group be similar (or related) to one another and different from (or unrelated to) the objects in other groups. The greater the similarity (or homogeneity) within a group and the greater the difference between groups, the better or more distinct the clustering.

In many applications, the notion of a cluster is not well defined. To better understand the difficulty of deciding what constitutes a cluster, consider Figure 8.1, which shows twenty points and three different ways of dividing them into clusters. The shapes of the markers indicate cluster membership. Figures 8.1(b) and 8.1(d) divide the data into two and six parts, respectively. However, the apparent division of each of the two larger clusters into three subclusters may simply be an artifact of the human visual system. Also, it may not be unreasonable to say that the points form four clusters, as shown in Figure 8.1(c). This figure illustrates that the definition of a cluster is imprecise and that the best definition depends on the nature of data and the desired results.

Cluster analysis is related to other techniques that are used to divide data objects into groups. For instance, clustering can be regarded as a form of classification in that it creates a labeling of objects with class (cluster) labels. However, it derives these labels only from the data. In contrast, classification



(a) Original points.

(b) Two clusters.

(c) Four clusters.

(d) Six clusters.

Figure 8.1. Different ways of clustering the same set of points.

in the sense of Chapter 4 is **supervised classification**; i.e., new, unlabeled objects are assigned a class label using a model developed from objects with known class labels. For this reason, cluster analysis is sometimes referred to as **unsupervised classification**. When the term classification is used without any qualification within data mining, it typically refers to supervised classification.

Also, while the terms **segmentation** and **partitioning** are sometimes used as synonyms for clustering, these terms are frequently used for approaches outside the traditional bounds of cluster analysis. For example, the term partitioning is often used in connection with techniques that divide graphs into subgraphs and that are not strongly connected to clustering. Segmentation often refers to the division of data into groups using simple techniques; e.g., an image can be split into segments based only on pixel intensity and color, or people can be divided into groups based on their income. Nonetheless, some work in graph partitioning and in image and market segmentation is related to cluster analysis.

8.1.2 Different Types of Clusterings

An entire collection of clusters is commonly referred to as a **clustering**, and in this section, we distinguish various types of clusterings: hierarchical (nested) versus partitional (unnested), exclusive versus overlapping versus fuzzy, and complete versus partial.

Hierarchical versus Partitional The most commonly discussed distinction among different types of clusterings is whether the set of clusters is nested or unnested, or in more traditional terminology, hierarchical or partitional. A **partitional clustering** is simply a division of the set of data objects into non-overlapping subsets (clusters) such that each data object is in exactly one subset. Taken individually, each collection of clusters in Figures 8.1 (b–d) is a partitional clustering.

If we permit clusters to have subclusters, then we obtain a **hierarchical clustering**, which is a set of nested clusters that are organized as a tree. Each node (cluster) in the tree (except for the leaf nodes) is the union of its children (subclusters), and the root of the tree is the cluster containing all the objects. Often, but not always, the leaves of the tree are singleton clusters of individual data objects. If we allow clusters to be nested, then one interpretation of Figure 8.1(a) is that it has two subclusters (Figure 8.1(b)), each of which, in turn, has three subclusters (Figure 8.1(d)). The clusters shown in Figures 8.1 (a–d), when taken in that order, also form a hierarchical (nested) clustering with, respectively, 1, 2, 4, and 6 clusters on each level. Finally, note that a hierarchical clustering can be viewed as a sequence of partitional clusterings and a partitional clustering can be obtained by taking any member of that sequence; i.e., by cutting the hierarchical tree at a particular level.

Exclusive versus Overlapping versus Fuzzy The clusterings shown in Figure 8.1 are all **exclusive**, as they assign each object to a single cluster. There are many situations in which a point could reasonably be placed in more than one cluster, and these situations are better addressed by non-exclusive clustering. In the most general sense, an **overlapping** or **non-exclusive clustering** is used to reflect the fact that an object can *simultaneously* belong to more than one group (class). For instance, a person at a university can be both an enrolled student and an employee of the university. A non-exclusive clustering is also often used when, for example, an object is “between” two or more clusters and could reasonably be assigned to any of these clusters. Imagine a point halfway between two of the clusters of Figure 8.1. Rather than make a somewhat arbitrary assignment of the object to a single cluster, it is placed in all of the “equally good” clusters.

In a **fuzzy clustering**, every object belongs to every cluster with a membership weight that is between 0 (absolutely doesn’t belong) and 1 (absolutely belongs). In other words, clusters are treated as fuzzy sets. (Mathematically, a fuzzy set is one in which an object belongs to any set with a weight that is between 0 and 1. In fuzzy clustering, we often impose the additional constraint that the sum of the weights for each object must equal 1.) Similarly, probabilistic clustering techniques compute the probability with which each point belongs to each cluster, and these probabilities must also sum to 1. Because the membership weights or probabilities for any object sum to 1, a fuzzy or probabilistic clustering does not address true multiclass situations, such as the case of a student employee, where an object belongs to multiple classes. Instead, these approaches are most appropriate for avoiding the arbitrariness of assigning an object to only one cluster when it may be close to several. In practice, a fuzzy or probabilistic clustering is often converted to an exclusive clustering by assigning each object to the cluster in which its membership weight or probability is highest.

Complete versus Partial A **complete clustering** assigns every object to a cluster, whereas a **partial clustering** does not. The motivation for a partial clustering is that some objects in a data set may not belong to well-defined groups. Many times objects in the data set may represent noise, outliers, or "uninteresting background." For example, some newspaper stories may share a common theme, such as global warming, while other stories are more generic or one-of-a-kind. Thus, to find the important topics in last month's stories, we may want to search only for clusters of documents that are tightly related by a common theme. In other cases, a complete clustering of the objects is desired. For example, an application that uses clustering to organize documents for browsing needs to guarantee that all documents can be browsed.

8.1.3 Different Types of Clusters

Clustering aims to find useful groups of objects (clusters), where usefulness is defined by the goals of the data analysis. Not surprisingly, there are several different notions of a cluster that prove useful in practice. In order to visually illustrate the differences among these types of clusters, we use two-dimensional points, as shown in Figure 8.2, as our data objects. We stress, however, that the types of clusters described here are equally valid for other kinds of data.

Well-Separated A cluster is a set of objects in which each object is closer (or more similar) to every other object in the cluster than to any object not in the cluster. Sometimes a threshold is used to specify that all the objects in a cluster must be sufficiently close (or similar) to one another. This idealistic definition of a cluster is satisfied only when the data contains natural clusters that are quite far from each other. Figure 8.2(a) gives an example of well-separated clusters that consists of two groups of points in a two-dimensional space. The distance between any two points in different groups is larger than the distance between any two points within a group. Well-separated clusters do not need to be globular, but can have any shape.

Prototype-Based A cluster is a set of objects in which each object is closer (more similar) to the prototype that defines the cluster than to the prototype of any other cluster. For data with continuous attributes, the prototype of a cluster is often a centroid, i.e., the average (mean) of all the points in the cluster. When a centroid is not meaningful, such as when the data has categorical attributes, the prototype is often a medoid, i.e., the most representative point of a cluster. For many types of data, the prototype can be regarded as the most central point, and in such instances, we commonly refer to prototype-based clusters as **center-based clusters**. Not surprisingly, such clusters tend to be globular. Figure 8.2(b) shows an example of center-based clusters.

Graph-Based If the data is represented as a graph, where the nodes are objects and the links represent connections among objects (see Section 2.1.2), then a cluster can be defined as a **connected component**; i.e., a group of objects that are connected to one another, but that have no connection to objects outside the group. An important example of graph-based clusters are **contiguity-based clusters**, where two objects are connected only if they are within a specified distance of each other. This implies that each object in a contiguity-based cluster is closer to some other object in the cluster than to any point in a different cluster. Figure 8.2(c) shows an example of such clusters for two-dimensional points. This definition of a cluster is useful when clusters are irregular or intertwined, but can have trouble when noise is present since, as illustrated by the two spherical clusters of Figure 8.2(c), a small bridge of points can merge two distinct clusters.

Other types of graph-based clusters are also possible. One such approach (Section 8.3.2) defines a cluster as a **clique**; i.e., a set of nodes in a graph that are completely connected to each other. Specifically, if we add connections between objects in the order of their distance from one another, a cluster is formed when a set of objects forms a clique. Like prototype-based clusters, such clusters tend to be globular.

Density-Based A cluster is a dense region of objects that is surrounded by a region of low density. Figure 8.2(d) shows some density-based clusters for data created by adding noise to the data of Figure 8.2(c). The two circular clusters are not merged, as in Figure 8.2(c), because the bridge between them fades into the noise. Likewise, the curve that is present in Figure 8.2(c) also fades into the noise and does not form a cluster in Figure 8.2(d). A density-based definition of a cluster is often employed when the clusters are irregular or intertwined, and when noise and outliers are present. By contrast, a contiguity-based definition of a cluster would not work well for the data of Figure 8.2(d) since the noise would tend to form bridges between clusters.

Shared-Property (Conceptual Clusters) More generally, we can define a cluster as a set of objects that share some property. This definition encompasses all the previous definitions of a cluster; e.g., objects in a center-based cluster share the property that they are all closest to the same centroid or medoid. However, the shared-property approach also includes new types of clusters. Consider the clusters shown in Figure 8.2(e). A triangular area (cluster) is adjacent to a rectangular one, and there are two intertwined circles (clusters). In both cases, a clustering algorithm would need a very specific concept of a cluster to successfully detect these clusters. The process of finding such clusters is called conceptual clustering. However, too sophisticated a notion of a cluster would take us into the area of pattern recognition, and thus, we only consider simpler types of clusters in this book.

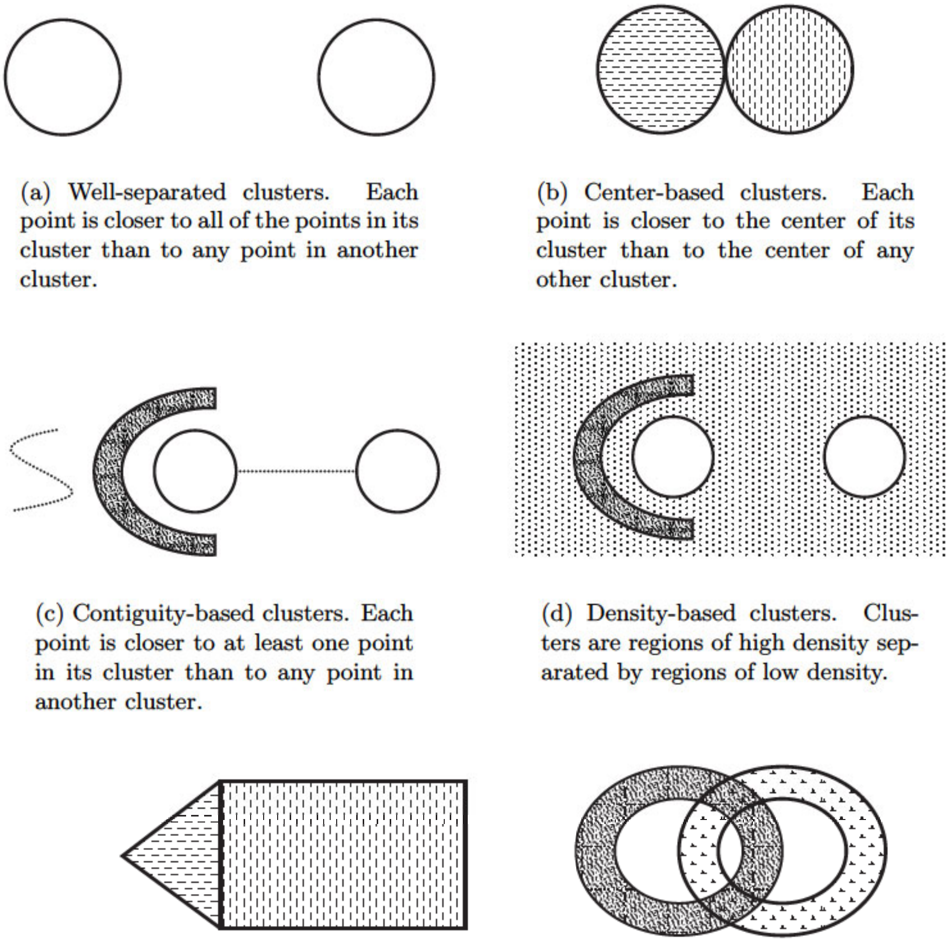
Road Map

In this chapter, we use the following three simple, but important techniques to introduce many of the concepts involved in cluster analysis.

- **K-means**. This is a prototype-based, partitional clustering technique that attempts to find a user-specified number of clusters (K), which are represented by their centroids.

- **Agglomerative Hierarchical Clustering.** This clustering approach refers to a collection of closely related clustering techniques that produce a hierarchical clustering by starting with each point as a singleton cluster and then repeatedly merging the two closest clusters until a single, all-encompassing cluster remains. Some of these techniques have a natural interpretation in terms of graph-based clustering, while others have an interpretation in terms of a prototype-based approach.

- **DBSCAN**. This is a density-based clustering algorithm that produces a partitional clustering, in which the number of clusters is automatically determined by the algorithm. Points in low-density regions are classified as noise and omitted; thus, DBSCAN does not produce a complete clustering.



(a) Well-separated clusters. Each point is closer to all of the points in its cluster than to any point in another cluster.

(b) Center-based clusters. Each point is closer to the center of its cluster than to the center of any other cluster.

(c) Contiguity-based clusters. Each point is closer to at least one point in its cluster than to any point in another cluster.

(d) Density-based clusters. Clusters are regions of high density separated by regions of low density.

(e) Conceptual clusters. Points in a cluster share some general property that derives from the entire set of points. (Points in the intersection of the circles belong to both.)

Figure 8.2. Different types of clusters as illustrated by sets of two-dimensional points.

8.2 K-means

Prototype-based clustering techniques create a one-level partitioning of the data objects. There are a number of such techniques, but two of the most prominent are K-means and K-medoid. K-means defines a prototype in terms of a centroid, which is usually the mean of a group of points, and is typically applied to objects in a continuous n-dimensional space. K-medoid defines a prototype in terms of a medoid, which is the most representative point for a group of points, and can be applied to a wide range of data since it requires only a proximity measure for a pair of objects. While a centroid almost never corresponds to an actual data point, a medoid, by its definition, must be an actual data point. In this section, we will focus solely on K-means, which is one of the oldest and most widely used clustering algorithms.

8.2.1 The Basic K-means Algorithm

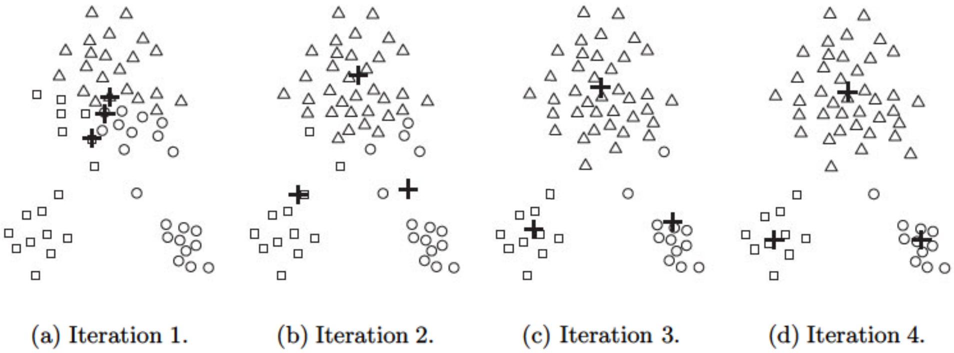
The K-means clustering technique is simple, and we begin with a description of the basic algorithm. We first choose K initial centroids, where K is a user-specified parameter, namely, the number of clusters desired. Each point is then assigned to the closest centroid, and each collection of points assigned to a centroid is a cluster. The centroid of each cluster is then updated based on the points assigned to the cluster. We repeat the assignment and update steps until no point changes clusters, or equivalently, until the centroids remain the same.

K-means is formally described by Algorithm 8.1. The operation of K-means is illustrated in Figure 8.3, which shows how, starting from three centroids, the final clusters are found in four assignment-update steps. In these and other figures displaying K-means clustering, each subfigure shows (1) the centroids at the start of the iteration and (2) the assignment of the points to those centroids. The centroids are indicated by the "+" symbol; all points belonging to the same cluster have the same marker shape.

Algorithm 8.1 Basic K-means algorithm.

```
Select $K$ points as initial centroids.
repeat
  Form $K$ clusters by assigning each point to its closest centroid.
  Recompute the centroid of each cluster.
until Centroids do not change.
```

In the first step, shown in Figure 8.3(a), points are assigned to the initial centroids, which are all in the larger group of points. For this example, we use the mean as the centroid. After points are assigned to a centroid, the centroid is then updated. Again, the figure for each step shows the centroid at the beginning of the step and the assignment of points to those centroids. In the second step, points are assigned to the updated centroids, and the centroids



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3. (d) Iteration 4.

Figure 8.3. Using the K-means algorithm to find three clusters in sample data.

are updated again. In steps 2, 3, and 4, which are shown in Figures 8.3 (b), (c), and (d), respectively, two of the centroids move to the two small groups of points at the bottom of the figures. When the K-means algorithm terminates in Figure 8.3(d), because no more changes occur, the centroids have identified the natural groupings of points.

For some combinations of proximity functions and types of centroids, K-means always converges to a solution; i.e., K-means reaches a state in which no points are shifting from one cluster to another, and hence, the centroids don't change. Because most of the convergence occurs in the early steps, however, the condition on line 5 of Algorithm 8.1 is often replaced by a weaker condition, e.g., repeat until only 1% of the points change clusters.

We consider each of the steps in the basic K-means algorithm in more detail and then provide an analysis of the algorithm's space and time complexity.

Assigning Points to the Closest Centroid

To assign a point to the closest centroid, we need a proximity measure that quantifies the notion of "closest" for the specific data under consideration. Euclidean (L_2) distance is often used for data points in Euclidean space, while cosine similarity is more appropriate for documents. However, there may be several types of proximity measures that are appropriate for a given type of data. For example, Manhattan (L_1) distance can be used for Euclidean data, while the Jaccard measure is often employed for documents.

Usually, the similarity measures used for K-means are relatively simple since the algorithm repeatedly calculates the similarity of each point to each centroid. In some cases, however, such as when the data is in low-dimensional

Table 8.1. Table of notation.

Symbol	Description
$\mathbf{x}$	An object.
C_i	The i^{th} cluster.
$\mathbf{c}_i$	The centroid of cluster C_i.
$\mathbf{c}$	The centroid of all points.
m_i	The number of objects in the i^{th} cluster.
m	The number of objects in the data set.
K	The number of clusters.

Euclidean space, it is possible to avoid computing many of the similarities, thus significantly speeding up the K-means algorithm. Bisecting K-means (described in Section 8.2.3) is another approach that speeds up K-means by reducing the number of similarities computed.

Centroids and Objective Functions

Step 4 of the K-means algorithm was stated rather generally as "recompute the centroid of each cluster," since the centroid can vary, depending on the proximity measure for the data and the goal of the clustering. The goal of the clustering is typically expressed by an objective function that depends on the proximities of the points to one another or to the cluster centroids; e.g., minimize the squared distance of each point to its closest centroid. We illustrate this with two examples. However, the key point is this: once we have specified a proximity measure and an objective function, the centroid that we should choose can often be determined mathematically. We provide mathematical details in Section 8.2.6, and provide a non-mathematical discussion of this observation here.

Data in Euclidean Space Consider data whose proximity measure is Euclidean distance. For our objective function, which measures the quality of a clustering, we use the **sum of the squared error (SSE)**, which is also known as scatter. In other words, we calculate the error of each data point, i.e., its Euclidean distance to the closest centroid, and then compute the total sum of the squared errors. Given two different sets of clusters that are produced by two different runs of K-means, we prefer the one with the smallest squared error since this means that the prototypes (centroids) of this clustering are a better representation of the points in their cluster. Using the notation in Table 8.1, the SSE is formally defined as follows:

$$\text{SSE} = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} dist(\mathbf{c}_i, \mathbf{x})^2 \tag{8.1}$$

where *dist* is the standard Euclidean (L_2) distance between two objects in Euclidean space.

Given these assumptions, it can be shown (see Section 8.2.6) that the centroid that minimizes the SSE of the cluster is the mean. Using the notation in Table 8.1, the centroid (mean) of the i^{th} cluster is defined by Equation 8.2.

$$\mathbf{c}_i = \frac{1}{m_i} \sum_{\mathbf{x} \in C_i} \mathbf{x} \tag{8.2}$$

To illustrate, the centroid of a cluster containing the three two-dimensional points, (1,1), (2,3), and (6,2), is $((1+2+6)/3, ((1+3+2)/3) = (3,2)$.

Steps 3 and 4 of the K-means algorithm directly attempt to minimize the SSE (or more generally, the objective function). Step 3 forms clusters by assigning points to their nearest centroid, which minimizes the SSE for the given set of centroids. Step 4 recomputes the centroids so as to further minimize the SSE. However, the actions of K-means in Steps 3 and 4 are only guaranteed to find a local minimum with respect to the SSE since they are based on optimizing the SSE for specific choices of the centroids and clusters, rather than for all possible choices. We will later see an example in which this leads to a suboptimal clustering.

Document Data To illustrate that K-means is not restricted to data in Euclidean space, we consider document data and the cosine similarity measure. Here we assume that the document data is represented as a document-term matrix as described on page 31. Our objective is to maximize the similarity of the documents in a cluster to the cluster centroid; this quantity is known as the **cohesion** of the cluster. For this objective it can be shown that the cluster centroid is, as for Euclidean data, the mean. The analogous quantity to the total SSE is the total cohesion, which is given by Equation 8.3.

$$\text{Total Cohesion} = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} cosine(\mathbf{x}, \mathbf{c}_i) \tag{8.3}$$

The General Case There are a number of choices for the proximity function, centroid, and objective function that can be used in the basic K-means

Table 8.2. K-means: Common choices for proximity, centroids, and objective functions.

Proximity Function	Centroid	Objective Function
Manhattan (L_1)	median	Minimize sum of the L_1 distance of an object to its cluster centroid
Squared Euclidean (L_2^2)	mean	Minimize sum of the squared L_2 distance of an object to its cluster centroid
cosine	mean	Maximize sum of the cosine similarity of an object to its cluster centroid
Bregman divergence	mean	Minimize sum of the Bregman divergence of an object to its cluster centroid

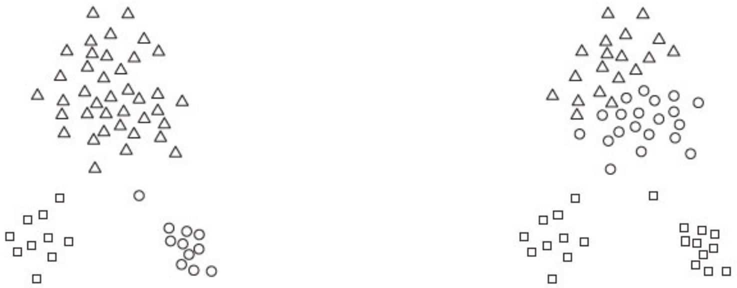
algorithm and that are guaranteed to converge. Table 8.2 shows some possible choices, including the two that we have just discussed. Notice that for Manhattan (L_1) distance and the objective of minimizing the sum of the distances, the appropriate centroid is the median of the points in a cluster.

The last entry in the table, Bregman divergence (Section 2.4.5), is actually a class of proximity measures that includes the squared Euclidean distance, L_2^2, the Mahalanobis distance, and cosine similarity. The importance of Bregman divergence functions is that any such function can be used as the basis of a K-means style clustering algorithm with the mean as the centroid. Specifically, if we use a Bregman divergence as our proximity function, then the resulting clustering algorithm has the usual properties of K-means with respect to convergence, local minima, etc. Furthermore, the properties of such a clustering algorithm can be developed for all possible Bregman divergences. Indeed, K-means algorithms that use cosine similarity or squared Euclidean distance are particular instances of a general clustering algorithm based on Bregman divergences.

For the rest our K-means discussion, we use two-dimensional data since it is easy to explain K-means and its properties for this type of data. But, as suggested by the last few paragraphs, K-means is a very general clustering algorithm and can be used with a wide variety of data types, such as documents and time series.

Choosing Initial Centroids

When random initialization of centroids is used, different runs of K-means typically produce different total SSEs. We illustrate this with the set of two-dimensional points shown in Figure 8.3, which has three natural clusters of points. Figure 8.4(a) shows a clustering solution that is the global minimum of



(a) Optimal clustering.

(b) Suboptimal clustering.

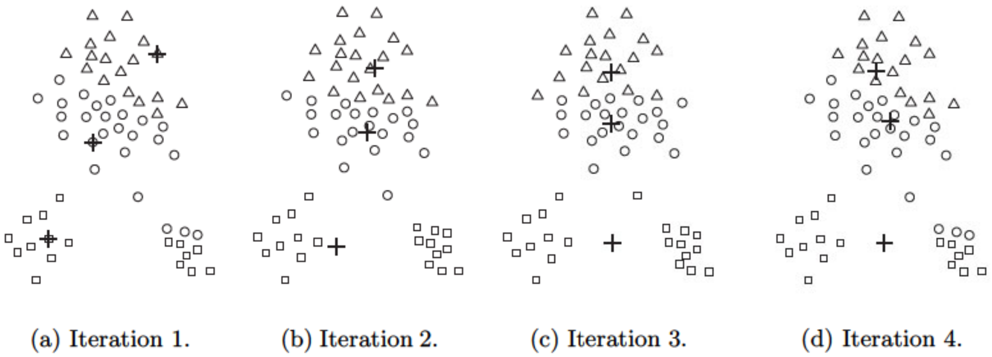
Figure 8.4. Three optimal and non-optimal clusters.

the SSE for three clusters, while Figure 8.4(b) shows a suboptimal clustering that is only a local minimum.

Choosing the proper initial centroids is the key step of the basic K-means procedure. A common approach is to choose the initial centroids randomly, but the resulting clusters are often poor.

Example 8.1 (Poor Initial Centroids). Randomly selected initial centroids may be poor. We provide an example of this using the same data set used in Figures 8.3 and 8.4. Figures 8.3 and 8.5 show the clusters that result from two particular choices of initial centroids. (For both figures, the positions of the cluster centroids in the various iterations are indicated by crosses.) In Figure 8.3, even though all the initial centroids are from one natural cluster, the minimum SSE clustering is still found. In Figure 8.5, however, even though the initial centroids seem to be better distributed, we obtain a suboptimal clustering, with higher squared error. ■

Example 8.2 (Limits of Random Initialization). One technique that is commonly used to address the problem of choosing initial centroids is to perform multiple runs, each with a different set of randomly chosen initial centroids, and then select the set of clusters with the minimum SSE. While simple, this strategy may not work very well, depending on the data set and the number of clusters sought. We demonstrate this using the sample data set shown in Figure 8.6(a). The data consists of two pairs of clusters, where the clusters in each (top-bottom) pair are closer to each other than to the clusters in the other pair. Figure 8.6 (b–d) shows that if we start with two initial centroids per pair of clusters, then even when both centroids are in a single



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3. (d) Iteration 4.

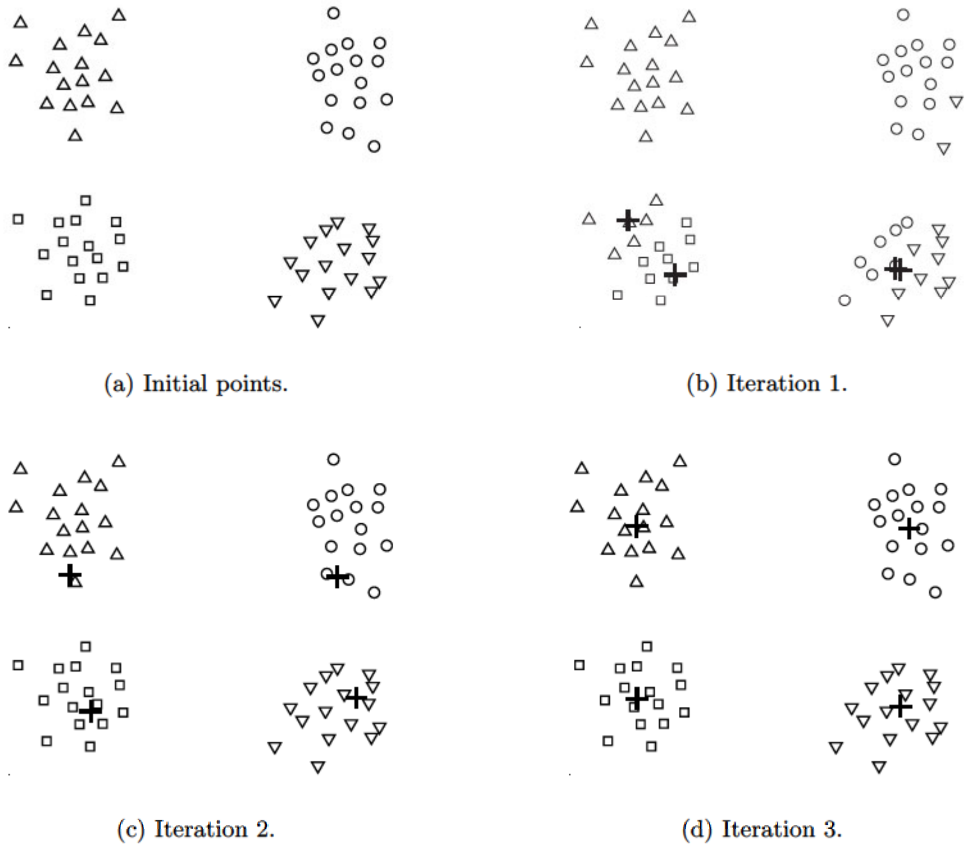
Figure 8.5. Poor starting centroids for K-means.

cluster, the centroids will redistribute themselves so that the "true" clusters are found. However, Figure 8.7 shows that if a pair of clusters has only one initial centroid and the other pair has three, then two of the true clusters will be combined and one true cluster will be split.

Note that an optimal clustering will be obtained as long as two initial centroids fall anywhere in a pair of clusters, since the centroids will redistribute themselves, one to each cluster. Unfortunately, as the number of clusters becomes larger, it is increasingly likely that at least one pair of clusters will have only one initial centroid. (See Exercise 4 on page 559.) In this case, because the pairs of clusters are farther apart than clusters within a pair, the K-means algorithm will not redistribute the centroids between pairs of clusters, and thus, only a local minimum will be achieved. ■

Because of the problems with using randomly selected initial centroids, which even repeated runs may not overcome, other techniques are often employed for initialization. One effective approach is to take a sample of points and cluster them using a hierarchical clustering technique. K clusters are extracted from the hierarchical clustering, and the centroids of those clusters are used as the initial centroids. This approach often works well, but is practical only if (1) the sample is relatively small, e.g., a few hundred to a few thousand (hierarchical clustering is expensive), and (2) K is relatively small compared to the sample size.

The following procedure is another approach to selecting initial centroids. Select the first point at random or take the centroid of all points. Then, for each successive initial centroid, select the point that is farthest from any of the initial centroids already selected. In this way, we obtain a set of initial


(a) Initial points.

(b) Iteration 1.

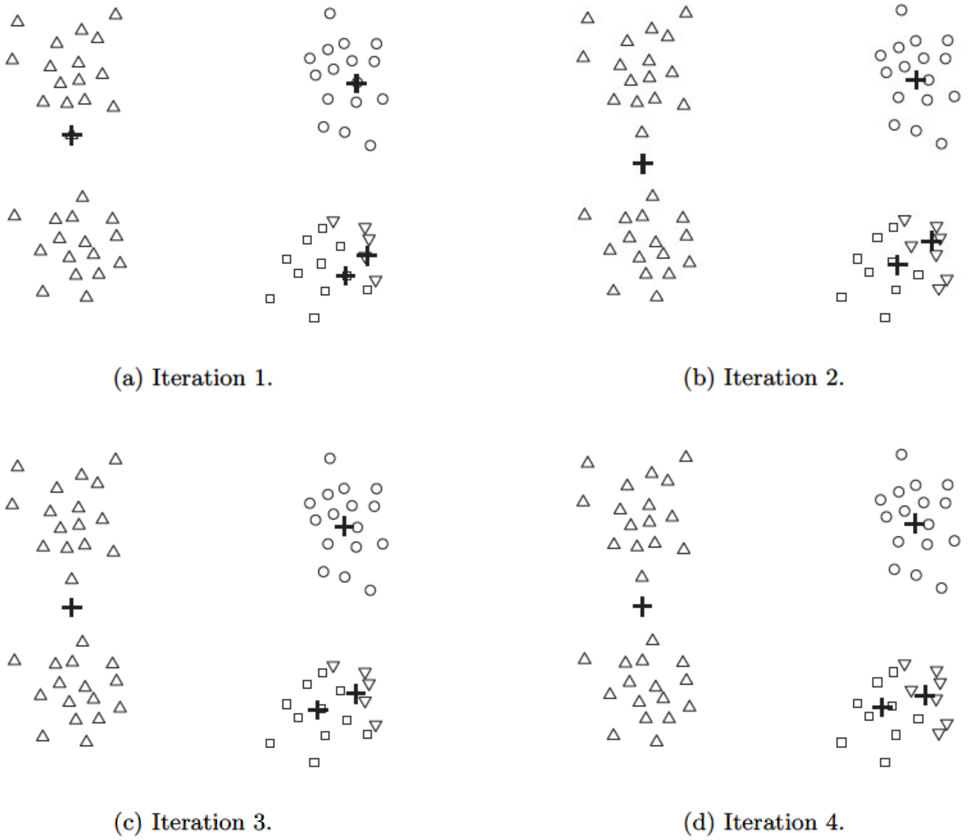
(c) Iteration 2.

(d) Iteration 3.

Figure 8.6. Two pairs of clusters with a pair of initial centroids within each pair of clusters.

centroids that is guaranteed to be not only randomly selected but also well separated. Unfortunately, such an approach can select outliers, rather than points in dense regions (clusters). Also, it is expensive to compute the farthest point from the current set of initial centroids. To overcome these problems, this approach is often applied to a sample of the points. Since outliers are rare, they tend not to show up in a random sample. In contrast, points from every dense region are likely to be included unless the sample size is very small. Also, the computation involved in finding the initial centroids is greatly reduced because the sample size is typically much smaller than the number of points.

Later on, we will discuss two other approaches that are useful for producing better-quality (lower SSE) clusterings: using a variant of K-means that


(a) Iteration 1.

(b) Iteration 2.

(c) Iteration 3.

(d) Iteration 4.

Figure 8.7. Two pairs of clusters with more or fewer than two initial centroids within a pair of clusters.

is less susceptible to initialization problems (bisecting K-means) and using postprocessing to "fixup" the set of clusters produced.

Time and Space Complexity

The space requirements for K-means are modest because only the data points and centroids are stored. Specifically, the storage required is $O((m + K)n)$, where m is the number of points and n is the number of attributes. The time requirements for K-means are also modest—basically linear in the number of data points. In particular, the time required is $O(I * K * m * n)$, where I is the number of iterations required for convergence. As mentioned, I is often small and can usually be safely bounded, as most changes typically occur in the first few iterations. Therefore, K-means is linear in m, the number of points, and is efficient as well as simple provided that K, the number of clusters, is significantly less than m.

8.2.2 K-means: Additional Issues

Handling Empty Clusters

One of the problems with the basic K-means algorithm given earlier is that empty clusters can be obtained if no points are allocated to a cluster during the assignment step. If this happens, then a strategy is needed to choose a replacement centroid, since otherwise, the squared error will be larger than necessary. One approach is to choose the point that is farthest away from any current centroid. If nothing else, this eliminates the point that currently contributes most to the total squared error. Another approach is to choose the replacement centroid from the cluster that has the highest SSE. This will typically split the cluster and reduce the overall SSE of the clustering. If there are several empty clusters, then this process can be repeated several times.

Outliers

When the squared error criterion is used, outliers can unduly influence the clusters that are found. In particular, when outliers are present, the resulting cluster centroids (prototypes) may not be as representative as they otherwise would be and thus, the SSE will be higher as well. Because of this, it is often useful to discover outliers and eliminate them beforehand. It is important, however, to appreciate that there are certain clustering applications for which outliers should not be eliminated. When clustering is used for data compression, every point must be clustered, and in some cases, such as financial analysis, apparent outliers, e.g., unusually profitable customers, can be the most interesting points.

An obvious issue is how to identify outliers. A number of techniques for identifying outliers will be discussed in Chapter 10. If we use approaches that remove outliers before clustering, we avoid clustering points that will not cluster well. Alternatively, outliers can also be identified in a postprocessing step. For instance, we can keep track of the SSE contributed by each point, and eliminate those points with unusually high contributions, especially over multiple runs. Also, we may want to eliminate small clusters since they frequently represent groups of outliers.

Reducing the SSE with Postprocessing

An obvious way to reduce the SSE is to find more clusters, i.e., to use a larger K. However, in many cases, we would like to improve the SSE, but don't want to increase the number of clusters. This is often possible because K-means typically converges to a local minimum. Various techniques are used to "fix up" the resulting clusters in order to produce a clustering that has lower SSE. The strategy is to focus on individual clusters since the total SSE is simply the sum of the SSE contributed by each cluster. (We will use the terminology *total SSE* and *cluster SSE*, respectively, to avoid any potential confusion.) We can change the total SSE by performing various operations on the clusters, such as splitting or merging clusters. One commonly used approach is to use alternate cluster splitting and merging phases. During a splitting phase, clusters are divided, while during a merging phase, clusters are combined. In this way, it is often possible to escape local SSE minima and still produce a clustering solution with the desired number of clusters. The following are some techniques used in the splitting and merging phases.

Two strategies that decrease the total SSE by increasing the number of clusters are the following:

Split a cluster: The cluster with the largest SSE is usually chosen, but we could also split the cluster with the largest standard deviation for one particular attribute.

Introduce a new cluster centroid: Often the point that is farthest from any cluster center is chosen. We can easily determine this if we keep track of the SSE contributed by each point. Another approach is to choose randomly from all points or from the points with the highest SSE.

Two strategies that decrease the number of clusters, while trying to minimize the increase in total SSE, are the following:

Disperse a cluster: This is accomplished by removing the centroid that corresponds to the cluster and reassigning the points to other clusters. Ideally, the cluster that is dispersed should be the one that increases the total SSE the least.

Merge two clusters: The clusters with the closest centroids are typically chosen, although another, perhaps better, approach is to merge the two clusters that result in the smallest increase in total SSE. These two merging strategies are the same ones that are used in the hierarchical clustering techniques known as the centroid method and Ward's method, respectively. Both methods are discussed in Section 8.3.

Updating Centroids Incrementally

Instead of updating cluster centroids after all points have been assigned to a cluster, the centroids can be updated incrementally, after each assignment of a point to a cluster. Notice that this requires either zero or two updates to cluster centroids at each step, since a point either moves to a new cluster (two updates) or stays in its current cluster (zero updates). Using an incremental update strategy guarantees that empty clusters are not produced since all clusters start with a single point, and if a cluster ever has only one point, then that point will always be reassigned to the same cluster.

In addition, if incremental updating is used, the relative weight of the point being added may be adjusted; e.g., the weight of points is often decreased as the clustering proceeds. While this can result in better accuracy and faster convergence, it can be difficult to make a good choice for the relative weight, especially in a wide variety of situations. These update issues are similar to those involved in updating weights for artificial neural networks.

Yet another benefit of incremental updates has to do with using objectives other than "minimize SSE." Suppose that we are given an arbitrary objective function to measure the goodness of a set of clusters. When we process an individual point, we can compute the value of the objective function for each possible cluster assignment, and then choose the one that optimizes the objective. Specific examples of alternative objective functions are given in Section 8.5.2.

On the negative side, updating centroids incrementally introduces an order dependency. In other words, the clusters produced may depend on the order in which the points are processed. Although this can be addressed by randomizing the order in which the points are processed, the basic K-means approach of updating the centroids after all points have been assigned to clusters has no order dependency. Also, incremental updates are slightly more expensive. However, K-means converges rather quickly, and therefore, the number of points switching clusters quickly becomes relatively small.

8.2.3 Bisecting K-means

The bisecting K-means algorithm is a straightforward extension of the basic K-means algorithm that is based on a simple idea: to obtain K clusters, split the set of all points into two clusters, select one of these clusters to split, and so on, until K clusters have been produced. The details of bisecting K-means are given by Algorithm 8.2.

Algorithm 8.2 Bisecting K-means algorithm.

```
Initialize the list of clusters to contain the cluster consisting of all points.
repeat
  Remove a cluster from the list of clusters.
  {Perform several "trial" bisections of the chosen cluster.}
  for $i = 1$ to $number\ of\ trials$ do
    Bisect the selected cluster using basic K-means.
  end for
  Select the two clusters from the bisection with the lowest total SSE.
  Add these two clusters to the list of clusters.
until Until the list of clusters contains $K$ clusters.
```

There are a number of different ways to choose which cluster to split. We can choose the largest cluster at each step, choose the one with the largest SSE, or use a criterion based on both size and SSE. Different choices result in different clusters.

We often refine the resulting clusters by using their centroids as the initial centroids for the basic K-means algorithm. This is necessary because, although the K-means algorithm is guaranteed to find a clustering that represents a local minimum with respect to the SSE, in bisecting K-means we are using the K-means algorithm "locally," i.e., to bisect individual clusters. Therefore, the final set of clusters does not represent a clustering that is a local minimum with respect to the total SSE.

Example 8.3 (Bisecting K-means and Initialization). To illustrate that bisecting K-means is less susceptible to initialization problems, we show, in Figure 8.8, how bisecting K-means finds four clusters in the data set originally shown in Figure 8.6(a). In iteration 1, two pairs of clusters are found; in iteration 2, the rightmost pair of clusters is split; and in iteration 3, the leftmost pair of clusters is split. Bisecting K-means has less trouble with initialization because it performs several trial bisections and takes the one with the lowest SSE, and because there are only two centroids at each step. ■

Finally, by recording the sequence of clusterings produced as K-means bisects clusters, we can also use bisecting K-means to produce a hierarchical clustering.

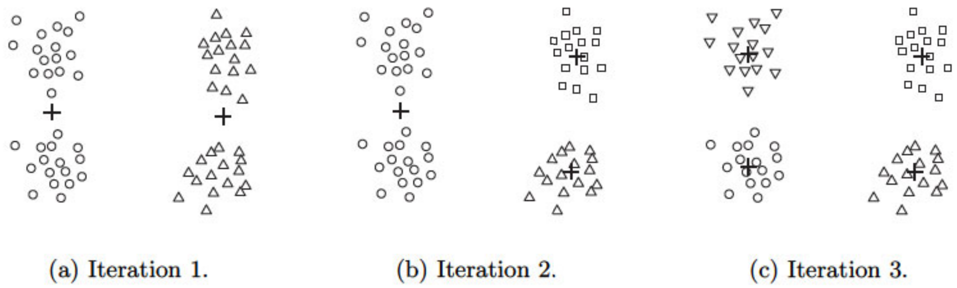


Figure 8.8. Bisecting K-means on the four clusters example.

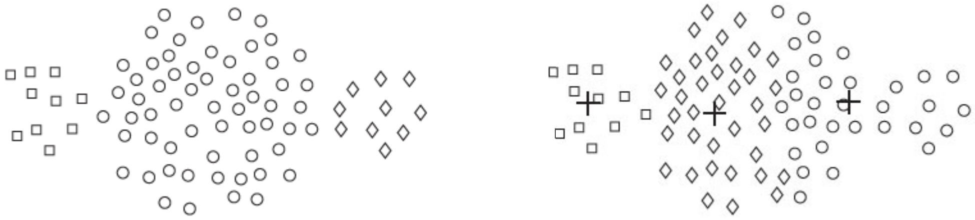
8.2.4 K-means and Different Types of Clusters

K-means and its variations have a number of limitations with respect to finding different types of clusters. In particular, K-means has difficulty detecting the "natural" clusters, when clusters have non-spherical shapes or widely different sizes or densities. This is illustrated by Figures 8.9, 8.10, and 8.11. In Figure 8.9, K-means cannot find the three natural clusters because one of the clusters is much larger than the other two, and hence, the larger cluster is broken, while one of the smaller clusters is combined with a portion of the larger cluster. In Figure 8.10, K-means fails to find the three natural clusters because the two smaller clusters are much denser than the larger cluster. Finally, in Figure 8.11, K-means finds two clusters that mix portions of the two natural clusters because the shape of the natural clusters is not globular.

The difficulty in these three situations is that the K-means objective function is a mismatch for the kinds of clusters we are trying to find since it is minimized by globular clusters of equal size and density or by clusters that are well separated. However, these limitations can be overcome, in some sense, if the user is willing to accept a clustering that breaks the natural clusters into a number of subclusters. Figure 8.12 shows what happens to the three previous data sets if we find six clusters instead of two or three. Each smaller cluster is pure in the sense that it contains only points from one of the natural clusters.

8.2.5 Strengths and Weaknesses

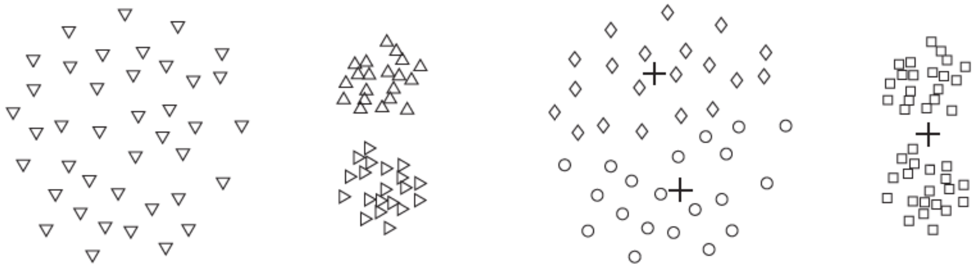
K-means is simple and can be used for a wide variety of data types. It is also quite efficient, even though multiple runs are often performed. Some variants, including bisecting K-means, are even more efficient, and are less susceptible to initialization problems. K-means is not suitable for all types of data,



(a) Original points.

(b) Three K-means clusters.

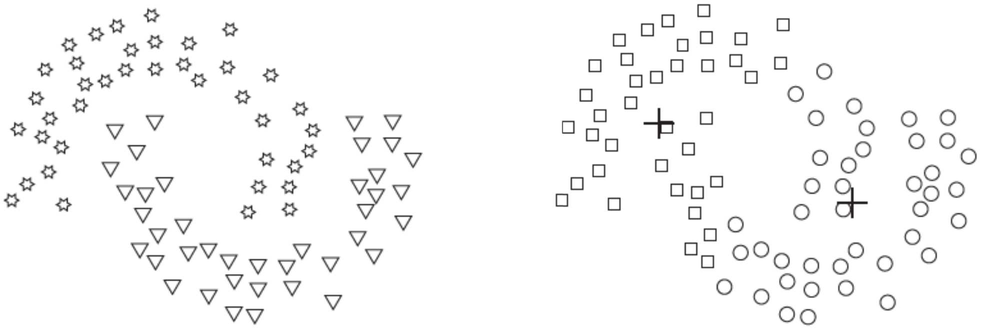
Figure 8.9. K-means with clusters of different size.



(a) Original points.

(b) Three K-means clusters.

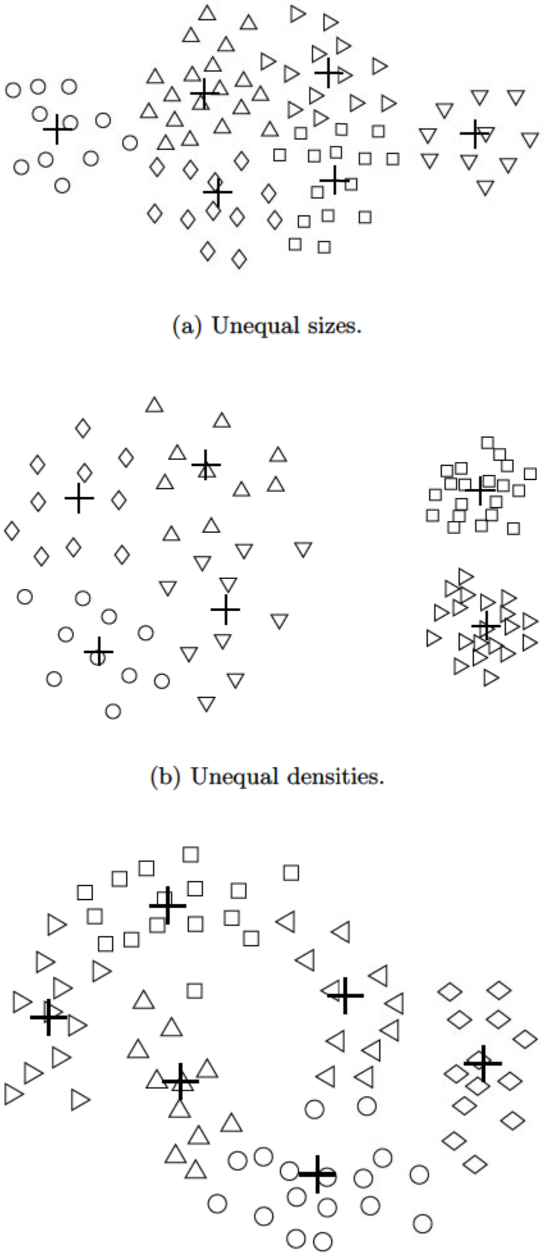
Figure 8.10. K-means with clusters of different density.



(a) Original points.

(b) Two K-means clusters.

Figure 8.11. K-means with non-globular clusters.



(a) Unequal sizes.

(b) Unequal densities.

(c) Non-spherical shapes.

Figure 8.12. Using K-means to find clusters that are subclusters of the natural clusters.

however. It cannot handle non-globular clusters or clusters of different sizes and densities, although it can typically find pure subclusters if a large enough number of clusters is specified. K-means also has trouble clustering data that contains outliers. Outlier detection and removal can help significantly in such situations. Finally, K-means is restricted to data for which there is a notion of a center (centroid). A related technique, K-medoid clustering, does not have this restriction, but is more expensive.

8.2.6 K-means as an Optimization Problem

Here, we delve into the mathematics behind K-means. This section, which can be skipped without loss of continuity, requires knowledge of calculus through partial derivatives. Familiarity with optimization techniques, especially those based on gradient descent, may also be helpful.

As mentioned earlier, given an objective function such as "minimize SSE," clustering can be treated as an optimization problem. One way to solve this problem—to find a global optimum—is to enumerate all possible ways of dividing the points into clusters and then choose the set of clusters that best satisfies the objective function, e.g., that minimizes the total SSE. Of course, this exhaustive strategy is computationally infeasible and as a result, a more practical approach is needed, even if such an approach finds solutions that are not guaranteed to be optimal. One technique, which is known as **gradient descent**, is based on picking an initial solution and then repeating the following two steps: compute the change to the solution that best optimizes the objective function and then update the solution.

We assume that the data is one-dimensional, i.e., $dist(x, y) = (x - y)^2$. This does not change anything essential, but greatly simplifies the notation.

Derivation of K-means as an Algorithm to Minimize the SSE

In this section, we show how the centroid for the K-means algorithm can be mathematically derived when the proximity function is Euclidean distance and the objective is to minimize the SSE. Specifically, we investigate how we can best update a cluster centroid so that the cluster SSE is minimized. In mathematical terms, we seek to minimize Equation 8.1, which we repeat here, specialized for one-dimensional data.

$$\text{SSE} = \sum_{i=1}^{K} \sum_{x \in C_i} (c_i - x)^2 \tag{8.4}$$

Here, C_i is the i^{th} cluster, x is a point in C_i, and c_i is the mean of the i^{th} cluster. See Table 8.1 for a complete list of notation.

We can solve for the k^{th} centroid c_k, which minimizes Equation 8.4, by differentiating the SSE, setting it equal to 0, and solving, as indicated below.

$$\begin{aligned}\frac{\partial}{\partial c_k}\text{SSE} &= \frac{\partial}{\partial c_k}\sum_{i=1}^{K}\sum_{x\in C_i}(c_i - x)^2 \\ &= \sum_{i=1}^{K}\sum_{x\in C_i}\frac{\partial}{\partial c_k}(c_i - x)^2 \\ &= \sum_{x\in C_k} 2*(c_k - x_k) = 0\end{aligned}$$

$$\sum_{x\in C_k} 2*(c_k - x_k) = 0 \Rightarrow m_k c_k = \sum_{x\in C_k} x_k \Rightarrow c_k = \frac{1}{m_k}\sum_{x\in C_k} x_k$$

Thus, as previously indicated, the best centroid for minimizing the SSE of a cluster is the mean of the points in the cluster.

Derivation of K-means for SAE

To demonstrate that the K-means algorithm can be applied to a variety of different objective functions, we consider how to partition the data into K clusters such that the sum of the Manhattan (L_1) distances of points from the center of their clusters is minimized. We are seeking to minimize the sum of the L_1 absolute errors (SAE) as given by the following equation, where $dist_{\text{L}_1}$ is the L_1 distance. Again, for notational simplicity, we use one-dimensional data, i.e., $dist_{\text{L}_1} = |c_i - x|$.

$$\text{SAE} = \sum_{i=1}^{K}\sum_{x\in C_i} dist_{\text{L}_1}(c_i, x) \qquad (8.5)$$

We can solve for the k^{th} centroid c_k, which minimizes Equation 8.5, by differentiating the SAE, setting it equal to 0, and solving.

$$\begin{aligned}\frac{\partial}{\partial c_k}\text{SAE} &= \frac{\partial}{\partial c_k}\sum_{i=1}^{K}\sum_{x\in C_i}|c_i - x| \\ &= \sum_{i=1}^{K}\sum_{x\in C_i}\frac{\partial}{\partial c_k}|c_i - x| \\ &= \sum_{x\in C_k}\frac{\partial}{\partial c_k}|c_k - x| = 0\end{aligned}$$

$$\sum_{x\in C_k}\frac{\partial}{\partial c_k}|c_k - x| = 0 \Rightarrow \sum_{x\in C_k} sign(x - c_k) = 0$$

If we solve for c_k, we find that $c_k = median\{x \in C_k\}$, the median of the points in the cluster. The median of a group of points is straightforward to compute and less susceptible to distortion by outliers.

8.3 Agglomerative Hierarchical Clustering

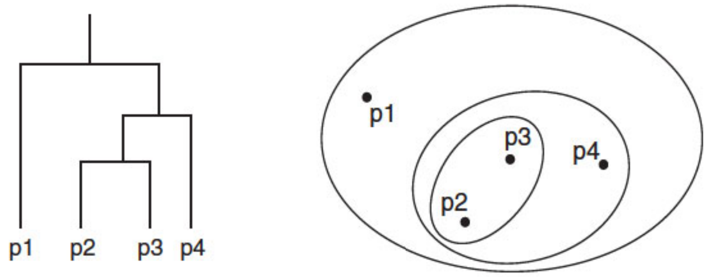
Hierarchical clustering techniques are a second important category of clustering methods. As with K-means, these approaches are relatively old compared to many clustering algorithms, but they still enjoy widespread use. There are two basic approaches for generating a hierarchical clustering:

Agglomerative: Start with the points as individual clusters and, at each step, merge the closest pair of clusters. This requires defining a notion of cluster proximity.

Divisive: Start with one, all-inclusive cluster and, at each step, split a cluster until only singleton clusters of individual points remain. In this case, we need to decide which cluster to split at each step and how to do the splitting.

Agglomerative hierarchical clustering techniques are by far the most common, and, in this section, we will focus exclusively on these methods. A divisive hierarchical clustering technique is described in Section 9.4.2.

A hierarchical clustering is often displayed graphically using a tree-like diagram called a **dendrogram**, which displays both the cluster-subcluster



(a) Dendrogram.

(b) Nested cluster diagram.

Figure 8.13. A hierarchical clustering of four points shown as a dendrogram and as nested clusters.

relationships and the order in which the clusters were merged (agglomerative view) or split (divisive view). For sets of two-dimensional points, such as those that we will use as examples, a hierarchical clustering can also be graphically represented using a nested cluster diagram. Figure 8.13 shows an example of these two types of figures for a set of four two-dimensional points. These points were clustered using the single-link technique that is described in Section 8.3.2.

8.3.1 Basic Agglomerative Hierarchical Clustering Algorithm

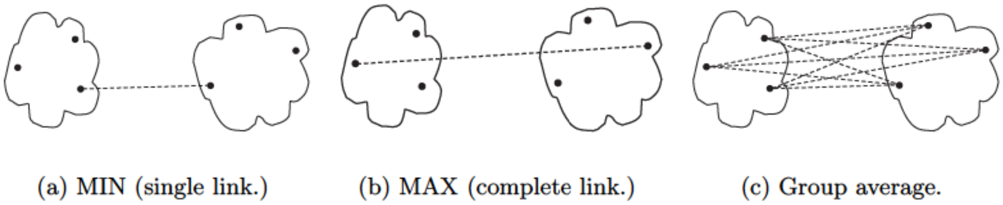
Many agglomerative hierarchical clustering techniques are variations on a single approach: starting with individual points as clusters, successively merge the two closest clusters until only one cluster remains. This approach is expressed more formally in Algorithm 8.3.

Algorithm 8.3 Basic agglomerative hierarchical clustering algorithm.

```
Compute the proximity matrix, if necessary.
repeat
    Merge the closest two clusters.
    Update the proximity matrix to reflect the proximity between the new
    cluster and the original clusters.
until Only one cluster remains.
```

Defining Proximity between Clusters

The key operation of Algorithm 8.3 is the computation of the proximity between two clusters, and it is the definition of cluster proximity that differentiates the various agglomerative hierarchical techniques that we will discuss. Cluster proximity is typically defined with a particular type of cluster in mind—see Section 8.1.2. For example, many agglomerative hierarchical clustering techniques, such as MIN, MAX, and Group Average, come from a graph-based view of clusters. **MIN** defines cluster proximity as the proximity between the closest two points that are in different clusters, or using graph terms, the shortest edge between two nodes in different subsets of nodes. This yields contiguity-based clusters as shown in Figure 8.2(c). Alternatively, **MAX** takes the proximity between the farthest two points in different clusters to be the cluster proximity, or using graph terms, the longest edge between two nodes in different subsets of nodes. (If our proximities are distances, then the names, MIN and MAX, are short and suggestive. For similarities, however, where higher values indicate closer points, the names seem reversed. For that reason, we usually prefer to use the alternative names, **single link** and **complete link**, respectively.) Another graph-based approach, the **group average** technique, defines cluster proximity to be the average pairwise proximities (average length of edges) of all pairs of points from different clusters. Figure 8.14 illustrates these three approaches.



(a) MIN (single link.) (b) MAX (complete link.) (c) Group average.

Figure 8.14. Graph-based definitions of cluster proximity

If, instead, we take a prototype-based view, in which each cluster is represented by a centroid, different definitions of cluster proximity are more natural. When using centroids, the cluster proximity is commonly defined as the proximity between cluster centroids. An alternative technique, **Ward's** method, also assumes that a cluster is represented by its centroid, but it measures the proximity between two clusters in terms of the increase in the SSE that results from merging the two clusters. Like K-means, Ward's method attempts to minimize the sum of the squared distances of points from their cluster centroids.

Time and Space Complexity

The basic agglomerative hierarchical clustering algorithm just presented uses a proximity matrix. This requires the storage of $\frac{1}{2}m^2$ proximities (assuming the proximity matrix is symmetric) where m is the number of data points. The space needed to keep track of the clusters is proportional to the number of clusters, which is $m-1$, excluding singleton clusters. Hence, the total space complexity is $O(m^2)$.

The analysis of the basic agglomerative hierarchical clustering algorithm is also straightforward with respect to computational complexity. $O(m^2)$ time is required to compute the proximity matrix. After that step, there are $m-1$ iterations involving steps 3 and 4 because there are m clusters at the start and two clusters are merged during each iteration. If performed as a linear search of the proximity matrix, then for the i^{th} iteration, step 3 requires $O((m-i+1)^2)$ time, which is proportional to the current number of clusters squared. Step 4 only requires $O(m-i+1)$ time to update the proximity matrix after the merger of two clusters. (A cluster merger affects only $O(m-i+1)$ proximities for the techniques that we consider.) Without modification, this would yield a time complexity of $O(m^3)$. If the distances from each cluster to all other clusters are stored as a sorted list (or heap), it is possible to reduce the cost of finding the two closest clusters to $O(m-i+1)$. However, because of the additional complexity of keeping data in a sorted list or heap, the overall time required for a hierarchical clustering based on Algorithm 8.3 is $O(m^2 \log m)$.

The space and time complexity of hierarchical clustering severely limits the size of data sets that can be processed. We discuss scalability approaches for clustering algorithms, including hierarchical clustering techniques, in Section 9.5.

8.3.2 Specific Techniques

Sample Data

To illustrate the behavior of the various hierarchical clustering algorithms, we shall use sample data that consists of 6 two-dimensional points, which are shown in Figure 8.15. The x and y coordinates of the points and the Euclidean distances between them are shown in Tables 8.3 and 8.4, respectively.

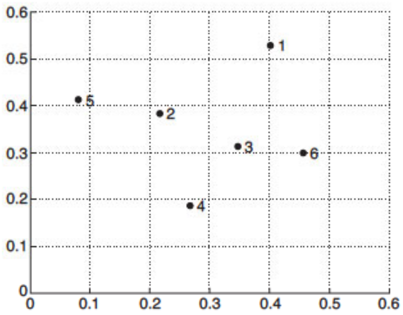


Figure 8.15. Set of 6 two-dimensional points.

Point	x Coordinate	y Coordinate
p1	0.40	0.53
p2	0.22	0.38
p3	0.35	0.32
p4	0.26	0.19
p5	0.08	0.41
p6	0.45	0.30

Table 8.3. xy coordinates of 6 points.

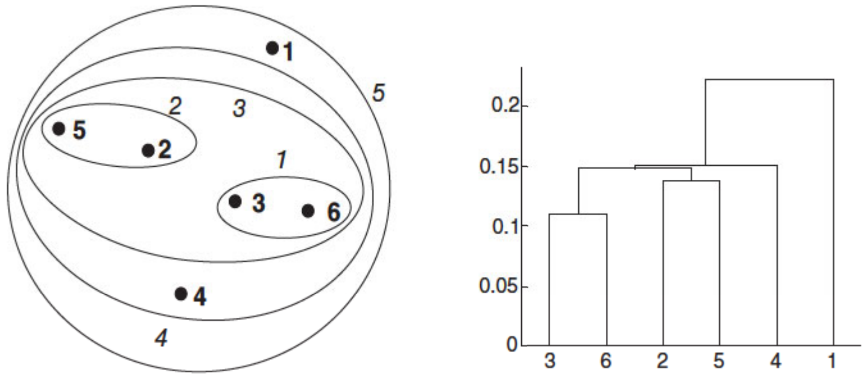
	p1	p2	p3	p4	p5	p6
p1	0.00	0.24	0.22	0.37	0.34	0.23
p2	0.24	0.00	0.15	0.20	0.14	0.25
p3	0.22	0.15	0.00	0.15	0.28	0.11
p4	0.37	0.20	0.15	0.00	0.29	0.22
p5	0.34	0.14	0.28	0.29	0.00	0.39
p6	0.23	0.25	0.11	0.22	0.39	0.00

Table 8.4. Euclidean distance matrix for 6 points.

Single Link or MIN

For the single link or MIN version of hierarchical clustering, the proximity of two clusters is defined as the minimum of the distance (maximum of the similarity) between any two points in the two different clusters. Using graph terminology, if you start with all points as singleton clusters and add links between points one at a time, shortest links first, then these single links combine the points into clusters. The single link technique is good at handling non-elliptical shapes, but is sensitive to noise and outliers.

Example 8.4 (Single Link). Figure 8.16 shows the result of applying the single link technique to our example data set of six points. Figure 8.16(a) shows the nested clusters as a sequence of nested ellipses, where the numbers associated with the ellipses indicate the order of the clustering. Figure 8.16(b) shows the same information, but as a dendrogram. The height at which two clusters are merged in the dendrogram reflects the distance of the two clusters. For instance, from Table 8.4, we see that the distance between points 3 and 6



(a) Single link clustering.

(b) Single link dendrogram.

Figure 8.16. Single link clustering of the six points shown in Figure 8.15.

is 0.11, and that is the height at which they are joined into one cluster in the dendrogram. As another example, the distance between clusters $\{3,6\}$ and $\{2,5\}$ is given by

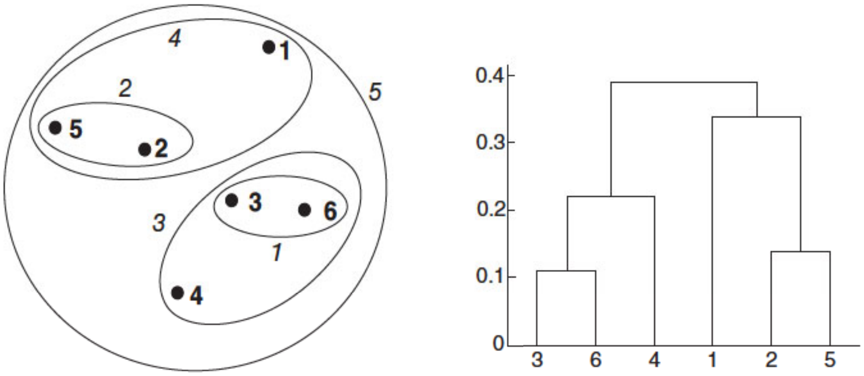
$$\begin{aligned} dist(\{3,6\},\{2,5\}) &= \min(dist(3,2), dist(6,2), dist(3,5), dist(6,5)) \\ &= \min(0.15, 0.25, 0.28, 0.39) \\ &= 0.15. \end{aligned}$$

■

Complete Link or MAX or CLIQUE

For the complete link or MAX version of hierarchical clustering, the proximity of two clusters is defined as the maximum of the distance (minimum of the similarity) between any two points in the two different clusters. Using graph terminology, if you start with all points as singleton clusters and add links between points one at a time, shortest links first, then a group of points is not a cluster until all the points in it are completely linked, i.e., form a *clique*. Complete link is less susceptible to noise and outliers, but it can break large clusters and it favors globular shapes.

Example 8.5 (Complete Link). Figure 8.17 shows the results of applying MAX to the sample data set of six points. As with single link, points 3 and 6



(a) Complete link clustering.

(b) Complete link dendrogram.

Figure 8.17. Complete link clustering of the six points shown in Figure 8.15.

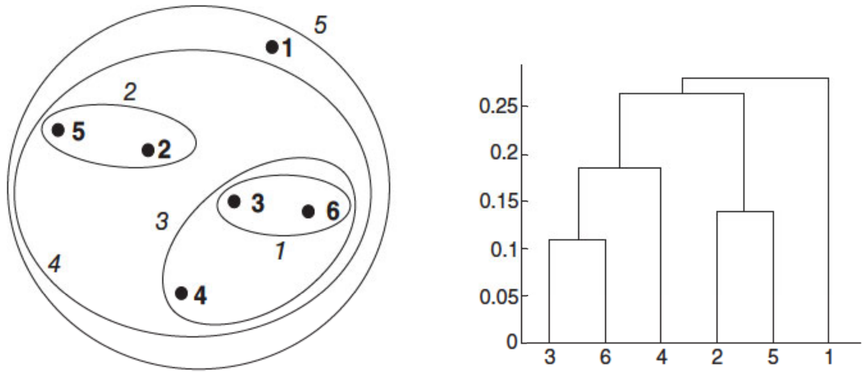
are merged first. However, $\{3, 6\}$ is merged with $\{4\}$, instead of $\{2, 5\}$ or $\{1\}$ because

$$\begin{aligned}
dist(\{3,6\},\{4\}) &= \max(dist(3,4), dist(6,4)) \\
&= \max(0.15, 0.22) \\
&= 0.22. \\
dist(\{3,6\},\{2,5\}) &= \max(dist(3,2), dist(6,2), dist(3,5), dist(6,5)) \\
&= \max(0.15, 0.25, 0.28, 0.39) \\
&= 0.39. \\
dist(\{3,6\},\{1\}) &= \max(dist(3,1), dist(6,1)) \\
&= \max(0.22, 0.23) \\
&= 0.23.
\end{aligned}$$

■

Group Average

For the group average version of hierarchical clustering, the proximity of two clusters is defined as the average pairwise proximity among all pairs of points in the different clusters. This is an intermediate approach between the single and complete link approaches. Thus, for group average, the cluster proxim-



(a) Group average clustering.

(b) Group average dendrogram.

Figure 8.18. Group average clustering of the six points shown in Figure 8.15.

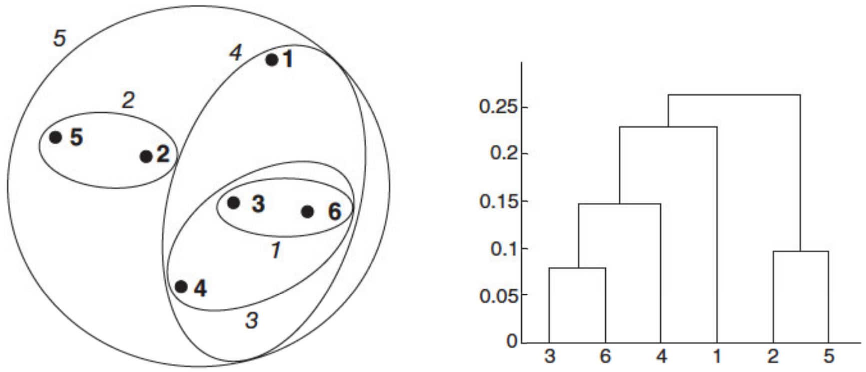
ity $proximity(C_i, C_j)$ of clusters C_i and C_j, which are of size m_i and m_j, respectively, is expressed by the following equation:

$$proximity(C_i, C_j) = \frac{\sum_{\substack{\mathbf{x} \in C_i \\ \mathbf{y} \in C_j}} proximity(\mathbf{x}, \mathbf{y})}{m_i * m_j}. \tag{8.6}$$

Example 8.6 (Group Average). Figure 8.18 shows the results of applying the group average approach to the sample data set of six points. To illustrate how group average works, we calculate the distance between some clusters.

$$\begin{aligned} dist(\{3,6,4\},\{1\}) &= (0.22 + 0.37 + 0.23)/(3 * 1) \\ &= 0.28 \\ dist(\{2,5\},\{1\}) &= (0.2357 + 0.3421)/(2 * 1) \\ &= 0.2889 \\ dist(\{3,6,4\},\{2,5\}) &= (0.15 + 0.28 + 0.25 + 0.39 + 0.20 + 0.29)/(6 * 2) \\ &= 0.26 \end{aligned}$$

Because $dist(\{3,6,4\},\{2,5\})$ is smaller than $dist(\{3,6,4\},\{1\})$ and $dist(\{2,5\},\{1\})$, clusters $\{3,6,4\}$ and $\{2,5\}$ are merged at the fourth stage. ■



(a) Ward's clustering.

(b) Ward's dendrogram.

Figure 8.19. Ward's clustering of the six points shown in Figure 8.15.

Ward's Method and Centroid Methods

For Ward's method, the proximity between two clusters is defined as the increase in the squared error that results when two clusters are merged. Thus, this method uses the same objective function as K-means clustering. While it may seem that this feature makes Ward's method somewhat distinct from other hierarchical techniques, it can be shown mathematically that Ward's method is very similar to the group average method when the proximity between two points is taken to be the square of the distance between them.

Example 8.7 (Ward's Method). Figure 8.19 shows the results of applying Ward's method to the sample data set of six points. The clustering that is produced is different from those produced by single link, complete link, and group average. ■

Centroid methods calculate the proximity between two clusters by calculating the distance between the centroids of clusters. These techniques may seem similar to K-means, but as we have remarked, Ward's method is the correct hierarchical analog.

Centroid methods also have a characteristic—often considered bad—that is not possessed by the other hierarchical clustering techniques that we have discussed: the possibility of **inversions**. Specifically, two clusters that are merged may be more similar (less distant) than the pair of clusters that were merged in a previous step. For the other methods, the distance between

Table 8.5. Table of Lance-Williams coefficients for common hierarchical clustering approaches.

Clustering Method	$\alpha_{\mathbf{A}}$	$\alpha_{\mathbf{B}}$	β	γ
Single Link	$1/2$	$1/2$	0	$-1/2$
Complete Link	$1/2$	$1/2$	0	$1/2$
Group Average	$\frac{m_A}{m_A+m_B}$	$\frac{m_B}{m_A+m_B}$	0	0
Centroid	$\frac{m_A}{m_A+m_B}$	$\frac{m_B}{m_A+m_B}$	$\frac{-m_A m_B}{(m_A+m_B)^2}$	0
Ward's	$\frac{m_A+m_Q}{m_A+m_B+m_Q}$	$\frac{m_B+m_Q}{m_A+m_B+m_Q}$	$\frac{-m_Q}{m_A+m_B+m_Q}$	0

merged clusters monotonically increases (or is, at worst, non-increasing) as we proceed from singleton clusters to one all-inclusive cluster.

8.3.3 The Lance-Williams Formula for Cluster Proximity

Any of the cluster proximities that we have discussed in this section can be viewed as a choice of different parameters (in the Lance-Williams formula shown below in Equation 8.7) for the proximity between clusters Q and R, where R is formed by merging clusters A and B. In this equation, $p(.,.)$ is a proximity function, while m_A, m_B, and m_Q are the number of points in clusters A, B, and Q, respectively. In other words, after we merge clusters A and B to form cluster R, the proximity of the new cluster, R, to an existing cluster, Q, is a linear function of the proximities of Q with respect to the original clusters A and B. Table 8.5 shows the values of these coefficients for the techniques that we have discussed.

$$p(R,Q) = \alpha_A \, p(A,Q) + \alpha_B \, p(B,Q) + \beta \, p(A,B) + \gamma \, |p(A,Q) - p(B,Q)| \quad (8.7)$$

Any hierarchical clustering technique that can be expressed using the Lance-Williams formula does not need to keep the original data points. Instead, the proximity matrix is updated as clustering occurs. While a general formula is appealing, especially for implementation, it is easier to understand the different hierarchical methods by looking directly at the definition of cluster proximity that each method uses.

8.3.4 Key Issues in Hierarchical Clustering

Lack of a Global Objective Function

We previously mentioned that agglomerative hierarchical clustering cannot be viewed as globally optimizing an objective function. Instead, agglomerative hierarchical clustering techniques use various criteria to decide locally, at each step, which clusters should be merged (or split for divisive approaches). This approach yields clustering algorithms that avoid the difficulty of attempting to solve a hard combinatorial optimization problem. (It can be shown that the general clustering problem for an objective function such as "minimize SSE" is computationally infeasible.) Furthermore, such approaches do not have problems with local minima or difficulties in choosing initial points. Of course, the time complexity of $O(m^2 \log m)$ and the space complexity of $O(m^2)$ are prohibitive in many cases.

Ability to Handle Different Cluster Sizes

One aspect of agglomerative hierarchical clustering that we have not yet discussed is how to treat the relative sizes of the pairs of clusters that are merged. (This discussion applies only to cluster proximity schemes that involve sums, such as centroid, Ward's, and group average.) There are two approaches: **weighted**, which treats all clusters equally, and **unweighted**, which takes the number of points in each cluster into account. Note that the terminology of weighted or unweighted refers to the data points, not the clusters. In other words, treating clusters of unequal size equally gives different weights to the points in different clusters, while taking the cluster size into account gives points in different clusters the same weight.

We will illustrate this using the group average technique discussed in Section 8.3.2, which is the unweighted version of the group average technique. In the clustering literature, the full name of this approach is the Unweighted Pair Group Method using Arithmetic averages (UPGMA). In Table 8.5, which gives the formula for updating cluster similarity, the coefficients for UPGMA involve the size of each of the clusters that were merged: $\alpha_A = \frac{m_A}{m_A+m_B}, \alpha_B = \frac{m_B}{m_A+m_B}, \beta = 0, \gamma = 0$. For the weighted version of group average—known as WPGMA—the coefficients are constants: $\alpha_A = 1/2, \alpha_B = 1/2, \beta = 0, \gamma = 0$. In general, unweighted approaches are preferred unless there is reason to believe that individual points should have different weights; e.g., perhaps classes of objects have been unevenly sampled.

Merging Decisions Are Final

Agglomerative hierarchical clustering algorithms tend to make good local decisions about combining two clusters since they can use information about the pairwise similarity of all points. However, once a decision is made to merge two clusters, it cannot be undone at a later time. This approach prevents a local optimization criterion from becoming a global optimization criterion.

For example, although the "minimize squared error" criterion from K-means is used in deciding which clusters to merge in Ward's method, the clusters at each level do not represent local minima with respect to the total SSE. Indeed, the clusters are not even stable, in the sense that a point in one cluster may be closer to the centroid of some other cluster than it is to the centroid of its current cluster. Nonetheless, Ward's method is often used as a robust method of initializing a K-means clustering, indicating that a local "minimize squared error" objective function does have a connection to a global "minimize squared error" objective function.

There are some techniques that attempt to overcome the limitation that merges are final. One approach attempts to fix up the hierarchical clustering by moving branches of the tree around so as to improve a global objective function. Another approach uses a partitional clustering technique such as K-means to create many small clusters, and then performs hierarchical clustering using these small clusters as the starting point.

8.3.5 Strengths and Weaknesses

The strengths and weakness of specific agglomerative hierarchical clustering algorithms were discussed above. More generally, such algorithms are typically used because the underlying application, e.g., creation of a taxonomy, requires a hierarchy. Also, there have been some studies that suggest that these algorithms can produce better-quality clusters. However, agglomerative hierarchical clustering algorithms are expensive in terms of their computational and storage requirements. The fact that all merges are final can also cause trouble for noisy, high-dimensional data, such as document data. In turn, these two problems can be addressed to some degree by first partially clustering the data using another technique, such as K-means.

8.4 DBSCAN

Density-based clustering locates regions of high density that are separated from one another by regions of low density. DBSCAN is a simple and effective density-based clustering algorithm that illustrates a number of important concepts that are important for any density-based clustering approach. In this section, we focus solely on DBSCAN after first considering the key notion of density. Other algorithms for finding density-based clusters are described in the next chapter.

8.4.1 Traditional Density: Center-Based Approach

Although there are not as many approaches for defining density as there are for defining similarity, there are several distinct methods. In this section we discuss the center-based approach on which DBSCAN is based. Other definitions of density will be presented in Chapter 9.

In the center-based approach, density is estimated for a particular point in the data set by counting the number of points within a specified radius, *Eps*, of that point. This includes the point itself. This technique is graphically illustrated by Figure 8.20. The number of points within a radius of *Eps* of point A is 7, including A itself.

This method is simple to implement, but the density of any point will depend on the specified radius. For instance, if the radius is large enough, then all points will have a density of m, the number of points in the data set. Likewise, if the radius is too small, then all points will have a density of 1. An approach for deciding on the appropriate radius for low-dimensional data is given in the next section in the context of our discussion of DBSCAN.

Classification of Points According to Center-Based Density

The center-based approach to density allows us to classify a point as being (1) in the interior of a dense region (a core point), (2) on the edge of a dense region (a border point), or (3) in a sparsely occupied region (a noise or background point). Figure 8.21 graphically illustrates the concepts of core, border, and noise points using a collection of two-dimensional points. The following text provides a more precise description.

Core points: These points are in the interior of a density-based cluster. A point is a core point if the number of points within a given neighborhood around the point as determined by the distance function and a user-specified distance parameter, *Eps*, exceeds a certain threshold, $MinPts$, which is also a user-specified parameter. In Figure 8.21, point A is a core point, for the indicated radius (*Eps*) if $MinPts \leq 7$.

Border points: A border point is not a core point, but falls within the neighborhood of a core point. In Figure 8.21, point B is a border point. A border point can fall within the neighborhoods of several core points.

Noise points: A noise point is any point that is neither a core point nor a border point. In Figure 8.21, point C is a noise point.

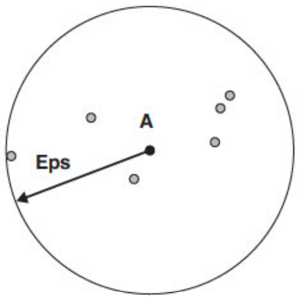


Figure 8.20. Center-based density.

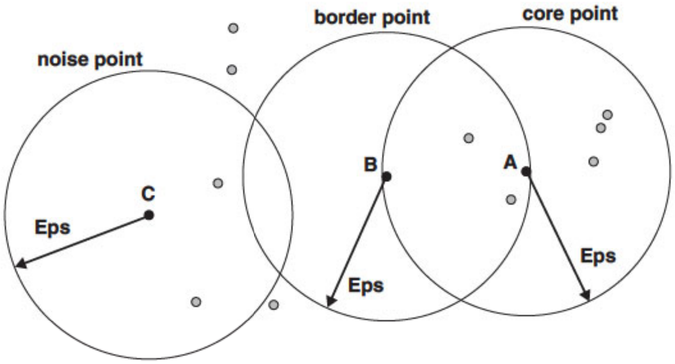


Figure 8.21. Core, border, and noise points.

8.4.2 The DBSCAN Algorithm

Given the previous definitions of core points, border points, and noise points, the DBSCAN algorithm can be informally described as follows. Any two core points that are close enough—within a distance *Eps* of one another—are put in the same cluster. Likewise, any border point that is close enough to a core point is put in the same cluster as the core point. (Ties may need to be resolved if a border point is close to core points from different clusters.) Noise points are discarded. The formal details are given in Algorithm 8.4. This algorithm uses the same concepts and finds the same clusters as the original DBSCAN, but is optimized for simplicity, not efficiency.

Algorithm 8.4 DBSCAN algorithm.

```
Label all points as core, border, or noise points.
Eliminate noise points.
Put an edge between all core points that are within $Eps$ of each other.
Make each group of connected core points into a separate cluster.
Assign each border point to one of the clusters of its associated core points.
```

Time and Space Complexity

The basic time complexity of the DBSCAN algorithm is $O(m \times$ time to find points in the *Eps*-neighborhood), where m is the number of points. In the worst case, this complexity is $O(m^2)$. However, in low-dimensional spaces, there are data structures, such as kd-trees, that allow efficient retrieval of all points within a given distance of a specified point, and the time complexity can be as low as $O(m \log m)$. The space requirement of DBSCAN, even for high-dimensional data, is $O(m)$ because it is only necessary to keep a small amount of data for each point, i.e., the cluster label and the identification of each point as a core, border, or noise point.

Selection of DBSCAN Parameters

There is, of course, the issue of how to determine the parameters *Eps* and $MinPts$. The basic approach is to look at the behavior of the distance from a point to its k^{th} nearest neighbor, which we will call the k-dist. For points that belong to some cluster, the value of k-dist will be small if k is not larger than the cluster size. Note that there will be some variation, depending on the density of the cluster and the random distribution of points, but on average, the range of variation will not be huge if the cluster densities are not radically different. However, for points that are not in a cluster, such as noise points, the k-dist will be relatively large. Therefore, if we compute the k-dist for all the data points for some k, sort them in increasing order, and then plot the sorted values, we expect to see a sharp change at the value of k-dist that corresponds to a suitable value of *Eps*. If we select this distance as the *Eps* parameter and take the value of k as the $MinPts$ parameter, then points for which k-dist is less than *Eps* will be labeled as core points, while other points will be labeled as noise or border points.

Figure 8.22 shows a sample data set, while the k-dist graph for the data is given in Figure 8.23. The value of *Eps* that is determined in this way depends on k, but does not change dramatically as k changes. If the value of k is too small, then even a small number of closely spaced points that are noise or outliers will be incorrectly labeled as clusters. If the value of k is too large, then small clusters (of size less than k) are likely to be labeled as noise. The original DBSCAN algorithm used a value of $k = 4$, which appears to be a reasonable value for most two-dimensional data sets.

Clusters of Varying Density

DBSCAN can have trouble with density if the density of clusters varies widely. Consider Figure 8.24, which shows four clusters embedded in noise. The density of the clusters and noise regions is indicated by their darkness. The noise around the pair of denser clusters, A and B, has the same density as clusters C and D. If the *Eps* threshold is low enough that DBSCAN finds C and D as clusters, then A and B and the points surrounding them will become a single

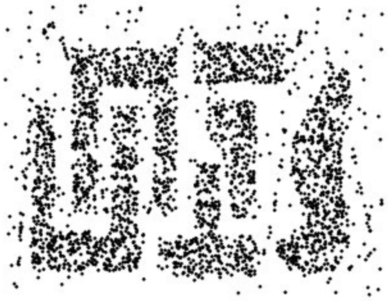


Figure 8.22. Sample data.

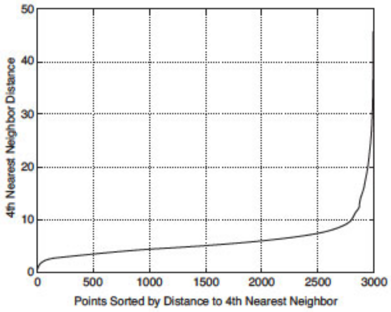


Figure 8.23. K-dist plot for sample data.

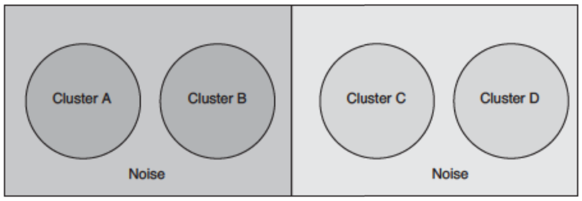


Figure 8.24. Four clusters embedded in noise.

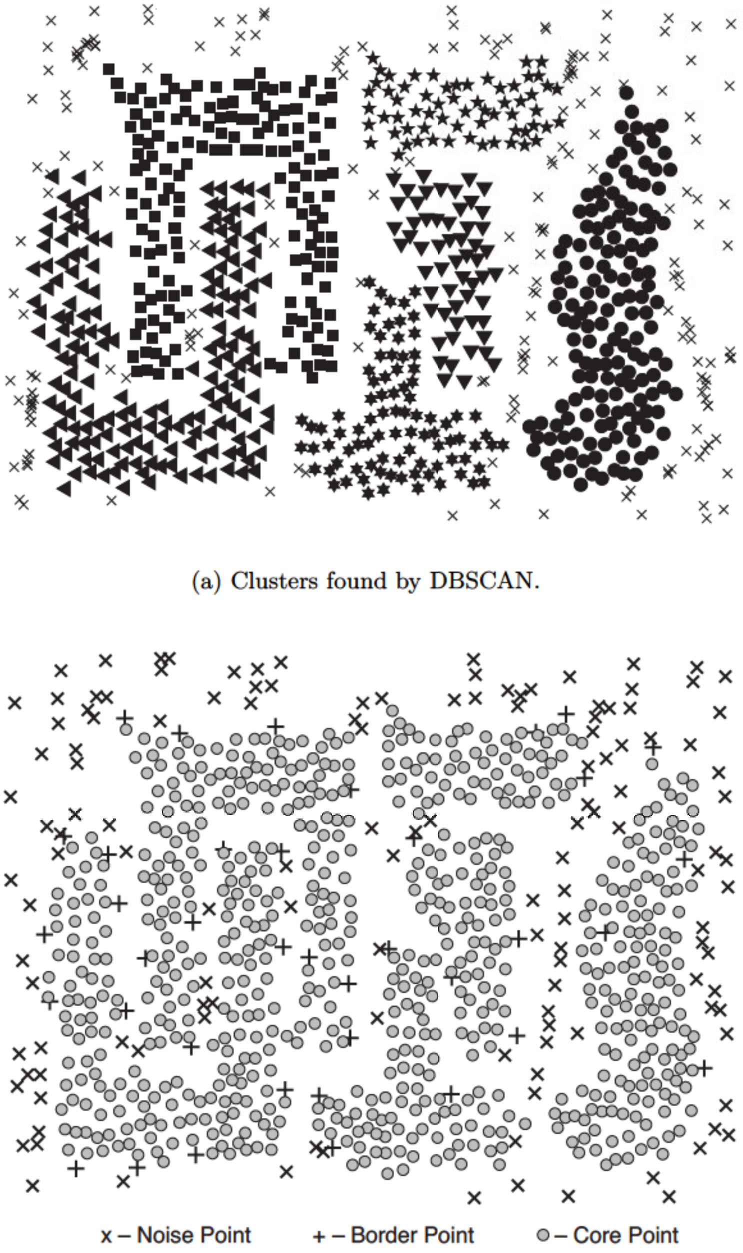
cluster. If the *Eps* threshold is high enough that DBSCAN finds A and B as separate clusters, and the points surrounding them are marked as noise, then C and D and the points surrounding them will also be marked as noise.

An Example

To illustrate the use of DBSCAN, we show the clusters that it finds in the relatively complicated two-dimensional data set shown in Figure 8.22. This data set consists of 3000 two-dimensional points. The *Eps* threshold for this data was found by plotting the sorted distances of the fourth nearest neighbor of each point (Figure 8.23) and identifying the value at which there is a sharp increase. We selected $Eps = 10$, which corresponds to the knee of the curve. The clusters found by DBSCAN using these parameters, i.e., $MinPts = 4$ and $Eps = 10$, are shown in Figure 8.25(a). The core points, border points, and noise points are displayed in Figure 8.25(b).

8.4.3 Strengths and Weaknesses

Because DBSCAN uses a density-based definition of a cluster, it is relatively resistant to noise and can handle clusters of arbitrary shapes and sizes. Thus,



(a) Clusters found by DBSCAN.

(b) Core, border, and noise points.

Figure 8.25. DBSCAN clustering of 3000 two-dimensional points.

DBSCAN can find many clusters that could not be found using K-means, such as those in Figure 8.22. As indicated previously, however, DBSCAN has trouble when the clusters have widely varying densities. It also has trouble with high-dimensional data because density is more difficult to define for such data. One possible approach to dealing with such issues is given in Section 9.4.8. Finally, DBSCAN can be expensive when the computation of nearest neighbors requires computing all pairwise proximities, as is usually the case for high-dimensional data.

8.5 Cluster Evaluation

In supervised classification, the evaluation of the resulting classification model is an integral part of the process of developing a classification model, and there are well-accepted evaluation measures and procedures, e.g., accuracy and cross-validation, respectively. However, because of its very nature, cluster evaluation is not a well-developed or commonly used part of cluster analysis. Nonetheless, cluster evaluation, or **cluster validation** as it is more traditionally called, is important, and this section will review some of the most common and easily applied approaches.

There might be some confusion as to why cluster evaluation is necessary. Many times, cluster analysis is conducted as a part of an exploratory data analysis. Hence, evaluation seems like an unnecessarily complicated addition to what is supposed to be an informal process. Furthermore, since there are a number of different types of clusters—in some sense, each clustering algorithm defines its own type of cluster—it may seem that each situation might require a different evaluation measure. For instance, K-means clusters might be evaluated in terms of the SSE, but for density-based clusters, which need not be globular, SSE would not work well at all.

Nonetheless, cluster evaluation should be a part of any cluster analysis. A key motivation is that almost every clustering algorithm will find clusters in a data set, even if that data set has no natural cluster structure. For instance, consider Figure 8.26, which shows the result of clustering 100 points that are randomly (uniformly) distributed on the unit square. The original points are shown in Figure 8.26(a), while the clusters found by DBSCAN, K-means, and complete link are shown in Figures 8.26(b), 8.26(c), and 8.26(d), respectively. Since DBSCAN found three clusters (after we set *Eps* by looking at the distances of the fourth nearest neighbors), we set K-means and complete link to find three clusters as well. (In Figure 8.26(b) the noise is shown by the small markers.) However, the clusters do not look compelling for any of the three methods. In higher dimensions, such problems cannot be so easily detected.

8.5.1 Overview

Being able to distinguish whether there is non-random structure in the data is just one important aspect of cluster validation. The following is a list of several important issues for cluster validation.

1. Determining the **clustering tendency** of a set of data, i.e., distinguishing whether non-random structure actually exists in the data.

2. Determining the correct number of clusters.

3. Evaluating how well the results of a cluster analysis fit the data *without* reference to external information.

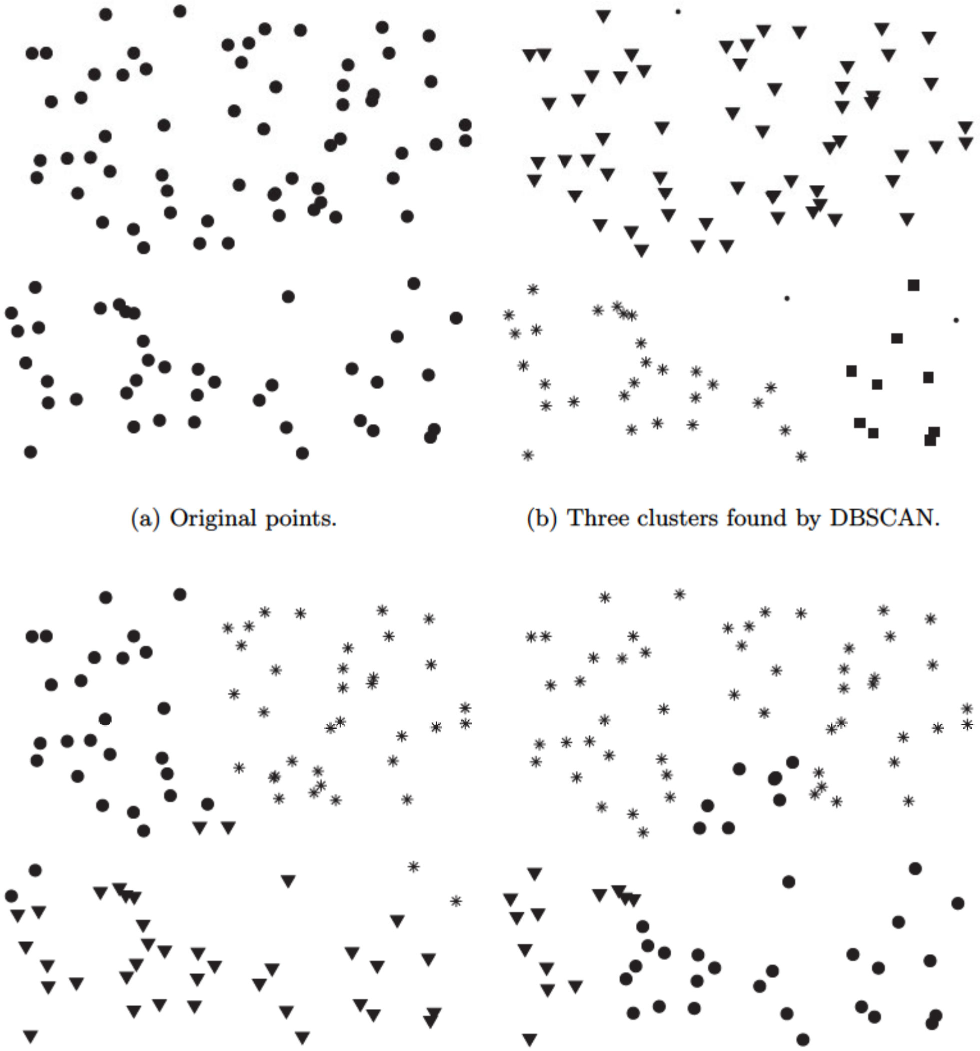
4. Comparing the results of a cluster analysis to externally known results, such as externally provided class labels.

5. Comparing two sets of clusters to determine which is better.

Notice that items 1, 2, and 3 do not make use of any external information—they are unsupervised techniques—while item 4 requires external information. Item 5 can be performed in either a supervised or an unsupervised manner. A further distinction can be made with respect to items 3, 4, and 5: Do we want to evaluate the entire clustering or just individual clusters?

While it is possible to develop various numerical measures to assess the different aspects of cluster validity mentioned above, there are a number of challenges. First, a measure of cluster validity may be quite limited in the scope of its applicability. For example, most work on measures of clustering tendency has been done for two- or three-dimensional spatial data. Second, we need a framework to interpret any measure. If we obtain a value of 10 for a measure that evaluates how well cluster labels match externally provided class labels, does this value represent a good, fair, or poor match? The goodness of a match often can be measured by looking at the statistical distribution of this value, i.e., how likely it is that such a value occurs by chance. Finally, if a measure is too complicated to apply or to understand, then few will use it.

The evaluation measures, or indices, that are applied to judge various aspects of cluster validity are traditionally classified into the following three types.



(a) Original points.

(b) Three clusters found by DBSCAN.

(c) Three clusters found by K-means.

(d) Three clusters found by complete link.

Figure 8.26. Clustering of 100 uniformly distributed points.

Unsupervised. Measures the goodness of a clustering structure without respect to external information. An example of this is the SSE. Unsupervised measures of cluster validity are often further divided into two classes: measures of **cluster cohesion** (compactness, tightness), which determine how closely related the objects in a cluster are, and measures of **cluster separation** (isolation), which determine how distinct or well-separated a cluster is from other clusters. Unsupervised measures are often called **internal indices** because they use only information present in the data set.

Supervised. Measures the extent to which the clustering structure discovered by a clustering algorithm matches some external structure. An example of a supervised index is entropy, which measures how well cluster labels match externally supplied class labels. Supervised measures are often called **external indices** because they use information not present in the data set.

Relative. Compares different clusterings or clusters. A relative cluster evaluation measure is a supervised or unsupervised evaluation measure that is used for the purpose of comparison. Thus, relative measures are not actually a separate type of cluster evaluation measure, but are instead a specific use of such measures. As an example, two K-means clusterings can be compared using either the SSE or entropy.

In the remainder of this section, we provide specific details concerning cluster validity. We first describe topics related to unsupervised cluster evaluation, beginning with (1) measures based on cohesion and separation, and (2) two techniques based on the proximity matrix. Since these approaches are useful only for partitional sets of clusters, we also describe the popular cophenetic correlation coefficient, which can be used for the unsupervised evaluation of a hierarchical clustering. We end our discussion of unsupervised evaluation with brief discussions about finding the correct number of clusters and evaluating clustering tendency. We then consider supervised approaches to cluster validity, such as entropy, purity, and the Jaccard measure. We conclude this section with a short discussion of how to interpret the values of (unsupervised or supervised) validity measures.

8.5.2 Unsupervised Cluster Evaluation Using Cohesion and Separation

Many internal measures of cluster validity for partitional clustering schemes are based on the notions of cohesion or separation. In this section, we use cluster validity measures for prototype- and graph-based clustering techniques to explore these notions in some detail. In the process, we will also see some interesting relationships between prototype- and graph-based clustering.

In general, we can consider expressing overall cluster validity for a set of K clusters as a weighted sum of the validity of individual clusters,

$$overall\ validity = \sum_{i=1}^{K} w_i\ validity(C_i). \tag{8.8}$$

The *validity* function can be cohesion, separation, or some combination of these quantities. The weights will vary depending on the cluster validity measure. In some cases, the weights are simply 1 or the size of the cluster, while in other cases they reflect a more complicated property, such as the square root of the cohesion. See Table 8.6. If the validity function is cohesion, then higher values are better. If it is separation, then lower values are better.

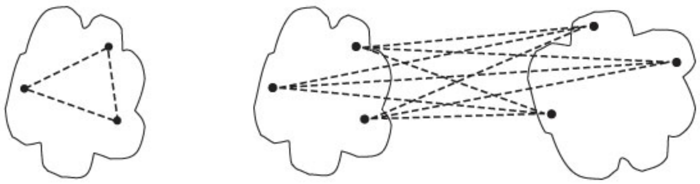
Graph-Based View of Cohesion and Separation

For graph-based clusters, the cohesion of a cluster can be defined as the sum of the weights of the links in the proximity graph that connect points within the cluster. See Figure 8.27(a). (Recall that the proximity graph has data objects as nodes, a link between each pair of data objects, and a weight assigned to each link that is the proximity between the two data objects connected by the link.) Likewise, the separation between two clusters can be measured by the sum of the weights of the links from points in one cluster to points in the other cluster. This is illustrated in Figure 8.27(b).

Mathematically, cohesion and separation for a graph-based cluster can be expressed using Equations 8.9 and 8.10, respectively. The *proximity* function can be a similarity, a dissimilarity, or a simple function of these quantities.

$$cohesion(C_i) = \sum_{\substack{\mathbf{x}\in C_i \\ \mathbf{y}\in C_i}} proximity(\mathbf{x}, \mathbf{y}) \tag{8.9}$$

$$separation(C_i, C_j) = \sum_{\substack{\mathbf{x}\in C_i \\ \mathbf{y}\in C_j}} proximity(\mathbf{x}, \mathbf{y}) \tag{8.10}$$



(a) Cohesion.

(b) Separation.

Figure 8.27. Graph-based view of cluster cohesion and separation.

Prototype-Based View of Cohesion and Separation

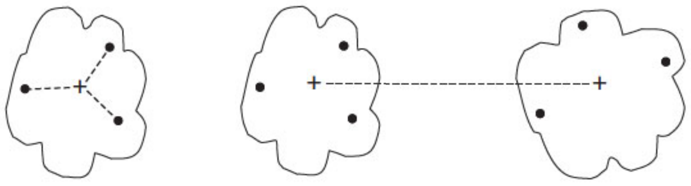
For prototype-based clusters, the cohesion of a cluster can be defined as the sum of the proximities with respect to the prototype (centroid or medoid) of the cluster. Similarly, the separation between two clusters can be measured by the proximity of the two cluster prototypes. This is illustrated in Figure 8.28, where the centroid of a cluster is indicated by a "+".

Cohesion for a prototype-based cluster is given in Equation 8.11, while two measures for separation are given in Equations 8.12 and 8.13, respectively, where $\mathbf{c}_i$ is the prototype (centroid) of cluster C_i and $\mathbf{c}$ is the overall prototype (centroid). There are two measures for separation because, as we will see shortly, the separation of cluster prototypes from an overall prototype is sometimes directly related to the separation of cluster prototypes from one another. Note that Equation 8.11 is the cluster SSE if we let proximity be the squared Euclidean distance.

$$\begin{aligned} cohesion(C_i) &= \sum_{\mathbf{x} \in C_i} proximity(\mathbf{x}, \mathbf{c}_i) && (8.11) \\ separation(C_i, C_j) &= proximity(\mathbf{c}_i, \mathbf{c}_j) && (8.12) \\ separation(C_i) &= proximity(\mathbf{c}_i, \mathbf{c}) && (8.13) \end{aligned}$$

Overall Measures of Cohesion and Separation

The previous definitions of cluster cohesion and separation gave us some simple and well-defined measures of cluster validity that can be combined into an overall measure of cluster validity by using a weighted sum, as indicated



(a) Cohesion. (b) Separation.

Figure 8.28. Prototype-based view of cluster cohesion and separation.

in Equation 8.8. However, we need to decide what weights to use. Not surprisingly, the weights used can vary widely, although typically they are some measure of cluster size.

Table 8.6 provides examples of validity measures based on cohesion and separation. $\mathcal{I}_1$ is a measure of cohesion in terms of the pairwise proximity of objects in the cluster divided by the cluster size. $\mathcal{I}_2$ is a measure of cohesion based on the sum of the proximities of objects in the cluster to the cluster centroid. $\mathcal{E}_1$ is a measure of separation defined as the proximity of a cluster centroid to the overall centroid multiplied by the number of objects in the cluster. $\mathcal{G}_1$, which is a measure based on both cohesion and separation, is the sum of the pairwise proximity of all objects in the cluster with all objects outside the cluster—the total weight of the edges of the proximity graph that must be cut to separate the cluster from all other clusters—divided by the sum of the pairwise proximity of objects in the cluster.

Table 8.6. Table of graph-based cluster evaluation measures.

Name	Cluster Measure	Cluster Weight	Type
$\mathcal{I}_1$	$\sum_{\substack{\mathbf{x}\in C_i \\ \mathbf{y}\in C_i}} \mathit{proximity}(\mathbf{x},\mathbf{y})$	$\frac{1}{m_i}$	graph-based cohesion
$\mathcal{I}_2$	$\sum_{\mathbf{x}\in C_i} \mathit{proximity}(\mathbf{x},\mathbf{c}_i)$	1	prototype-based cohesion
$\mathcal{E}_1$	$\mathit{proximity}(\mathbf{c}_i,\mathbf{c})$	m_i	prototype-based separation
$\mathcal{G}_1$	$\sum_{\substack{j=1 \\ j\neq i}}^{k} \sum_{\substack{\mathbf{x}\in C_i \\ \mathbf{y}\in C_j}} \mathit{proximity}(\mathbf{x},\mathbf{y})$	$\frac{1}{\sum_{\substack{\mathbf{x}\in C_i \\ \mathbf{y}\in C_i}} \mathit{proximity}(\mathbf{x},\mathbf{y})}$	graph-based separation and cohesion

Note that any unsupervised measure of cluster validity potentially can be used as an objective function for a clustering algorithm and vice versa. The CLUstering TOolkit (CLUTO) (see the bibliographic notes) uses the cluster evaluation measures described in Table 8.6, as well as some other evaluation measures not mentioned here, to drive the clustering process. It does this by using an algorithm that is similar to the incremental K-means algorithm discussed in Section 8.2.2. Specifically, each point is assigned to the cluster that produces the best value for the cluster evaluation function. The cluster evaluation measure $\mathcal{I}_2$ corresponds to traditional K-means and produces clusters that have good SSE values. The other measures produce clusters that are not as good with respect to SSE, but that are more optimal with respect to the specified cluster validity measure.

Relationship between Prototype-Based Cohesion and Graph-Based Cohesion

While the graph-based and prototype-based approaches to measuring the cohesion and separation of a cluster seem distinct, for some proximity measures they are equivalent. For instance, for the SSE and points in Euclidean space, it can be shown (Equation 8.14) that the average pairwise distance between the points in a cluster is equivalent to the SSE of the cluster. See Exercise 27 on page 566.

$$\text{Cluster SSE} = \sum_{\mathbf{x} \in C_i} dist(\mathbf{c}_i, \mathbf{x})^2 = \frac{1}{2m_i} \sum_{\mathbf{x} \in C_i} \sum_{\mathbf{y} \in C_i} dist(\mathbf{x}, \mathbf{y})^2 \qquad (8.14)$$

Two Approaches to Prototype-Based Separation

When proximity is measured by Euclidean distance, the traditional measure of separation between clusters is the between group sum of squares (SSB), which is the sum of the squared distance of a cluster centroid, $\mathbf{c}_i$, to the overall mean, $\mathbf{c}$, of all the data points. By summing the SSB over all clusters, we obtain the total SSB, which is given by Equation 8.15, where $\mathbf{c}_i$ is the mean of the i^{th} cluster and $\mathbf{c}$ is the overall mean. The higher the total SSB of a clustering, the more separated the clusters are from one another.

$$\text{Total SSB} = \sum_{i=1}^{K} m_i \, dist(\mathbf{c}_i, \mathbf{c})^2 \qquad (8.15)$$

It is straightforward to show that the total SSB is directly related to the pairwise distances between the centroids. In particular, if the cluster sizes are equal, i.e., $m_i = m/K$, then this relationship takes the simple form given by Equation 8.16. (See Exercise 28 on page 566.) It is this type of equivalence that motivates the definition of prototype separation in terms of both Equations 8.12 and 8.13.

$$\text{Total SSB} = \frac{1}{2K}\sum_{i=1}^{K}\sum_{j=1}^{K}\frac{m}{K}\ dist(\mathbf{c}_i, \mathbf{c}_j)^2 \tag{8.16}$$

Relationship between Cohesion and Separation

In some cases, there is also a strong relationship between cohesion and separation. Specifically, it is possible to show that the sum of the total SSE and the total SSB is a constant; i.e., that it is equal to the total sum of squares (TSS), which is the sum of squares of the distance of each point to the overall mean of the data. The importance of this result is that minimizing SSE (cohesion) is equivalent to maximizing SSB (separation).

We provide the proof of this fact below, since the approach illustrates techniques that are also applicable to proving the relationships stated in the last two sections. To simplify the notation, we assume that the data is one-dimensional, i.e., $dist(x, y) = (x-y)^2$. Also, we use the fact that the cross-term $\sum_{i=1}^{K}\sum_{x\in C_i}(x - c_i)(c - c_i)$ is 0. (See Exercise 29 on page 566.)

$$\begin{aligned}
\text{TSS} &= \sum_{i=1}^{K}\sum_{x\in C_i}(x-c)^2 \\
&= \sum_{i=1}^{K}\sum_{x\in C_i}((x-c_i)-(c-c_i))^2 \\
&= \sum_{i=1}^{K}\sum_{x\in C_i}(x-c_i)^2 - 2\sum_{i=1}^{K}\sum_{x\in C_i}(x-c_i)(c-c_i) + \sum_{i=1}^{K}\sum_{x\in C_i}(c-c_i)^2 \\
&= \sum_{i=1}^{K}\sum_{x\in C_i}(x-c_i)^2 + \sum_{i=1}^{K}\sum_{x\in C_i}(c-c_i)^2 \\
&= \sum_{i=1}^{K}\sum_{x\in C_i}(x-c_i)^2 + \sum_{i=1}^{K}|C_i|(c-c_i)^2 \\
&= \text{SSE} + \text{SSB}
\end{aligned}$$

Evaluating Individual Clusters and Objects

So far, we have focused on using cohesion and separation in the overall evaluation of a group of clusters. Many of these measures of cluster validity also can be used to evaluate individual clusters and objects. For example, we can rank individual clusters according to their specific value of cluster validity, i.e., cluster cohesion or separation. A cluster that has a high value of cohesion may be considered better than a cluster that has a lower value. This information often can be used to improve the quality of a clustering. If, for example, a cluster is not very cohesive, then we may want to split it into several subclusters. On the other hand, if two clusters are relatively cohesive, but not well separated, we may want to merge them into a single cluster.

We can also evaluate the objects within a cluster in terms of their contribution to the overall cohesion or separation of the cluster. Objects that contribute more to the cohesion and separation are near the "interior" of the cluster. Those objects for which the opposite is true are probably near the "edge" of the cluster. In the following section, we consider a cluster evaluation measure that uses an approach based on these ideas to evaluate points, clusters, and the entire set of clusters.

The Silhouette Coefficient

The popular method of silhouette coefficients combines both cohesion and separation. The following steps explain how to compute the silhouette coefficient for an individual point, a process that consists of the following three steps. We use distances, but an analogous approach can be used for similarities.

1. For the i^{th} object, calculate its average distance to all other objects in its cluster. Call this value a_i.
2. For the i^{th} object and any cluster not containing the object, calculate the object's average distance to all the objects in the given cluster. Find the minimum such value with respect to all clusters; call this value b_i.
3. For the i^{th} object, the silhouette coefficient is $s_i = (b_i - a_i)/\max(a_i, b_i)$.

The value of the silhouette coefficient can vary between -1 and 1. A negative value is undesirable because this corresponds to a case in which a_i, the average distance to points in the cluster, is greater than b_i, the minimum average distance to points in another cluster. We want the silhouette coefficient to be positive ($a_i < b_i$), and for a_i to be as close to 0 as possible, since the coefficient assumes its maximum value of 1 when $a_i = 0$.

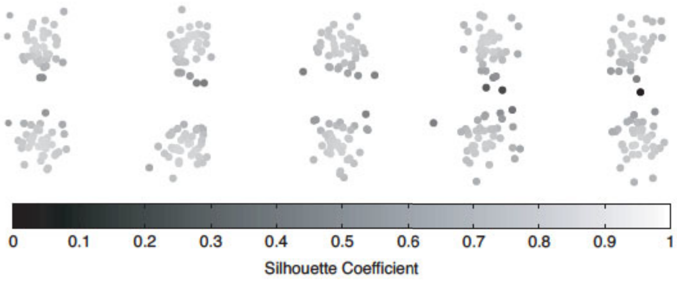


Figure 8.29. Silhouette coefficients for points in ten clusters.

We can compute the average silhouette coefficient of a cluster by simply taking the average of the silhouette coefficients of points belonging to the cluster. An overall measure of the goodness of a clustering can be obtained by computing the average silhouette coefficient of all points.

Example 8.8 (Silhouette Coefficient). Figure 8.29 shows a plot of the silhouette coefficients for points in 10 clusters. Darker shades indicate lower silhouette coefficients. ■

8.5.3 Unsupervised Cluster Evaluation Using the Proximity Matrix

In this section, we examine a couple of unsupervised approaches for assessing cluster validity that are based on the proximity matrix. The first compares an actual and idealized proximity matrix, while the second uses visualization.

Measuring Cluster Validity via Correlation

If we are given the similarity matrix for a data set and the cluster labels from a cluster analysis of the data set, then we can evaluate the "goodness" of the clustering by looking at the correlation between the similarity matrix and an ideal version of the similarity matrix based on the cluster labels. (With minor changes, the following applies to proximity matrices, but for simplicity, we discuss only similarity matrices.) More specifically, an ideal cluster is one whose points have a similarity of 1 to all points in the cluster, and a similarity of 0 to all points in other clusters. Thus, if we sort the rows and columns of the similarity matrix so that all objects belonging to the same class are together, then an ideal similarity matrix has a **block diagonal** structure. In other words, the similarity is non-zero, i.e., 1, inside the blocks of the similarity matrix whose entries represent intra-cluster similarity, and 0 elsewhere. The ideal similarity matrix is constructed by creating a matrix that has one row and one column for each data point—just like an actual similarity matrix—and assigning a 1 to an entry if the associated pair of points belongs to the same cluster. All other entries are 0.

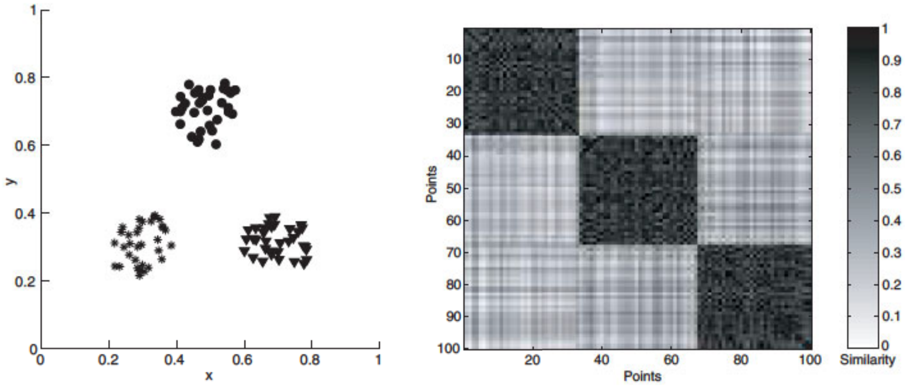
High correlation between the ideal and actual similarity matrices indicates that the points that belong to the same cluster are close to each other, while low correlation indicates the opposite. (Since the actual and ideal similarity matrices are symmetric, the correlation is calculated only among the $n(n-1)/2$ entries below or above the diagonal of the matrices.) Consequently, this is not a good measure for many density- or contiguity-based clusters, because they are not globular and may be closely intertwined with other clusters.

Example 8.9 (Correlation of Actual and Ideal Similarity Matrices). To illustrate this measure, we calculated the correlation between the ideal and actual similarity matrices for the K-means clusters shown in Figure 8.26(c) (random data) and Figure 8.30(a) (data with three well-separated clusters). The correlations were 0.5810 and 0.9235, respectively, which reflects the expected result that the clusters found by K-means in the random data are worse than the clusters found by K-means in data with well-separated clusters. ■

Judging a Clustering Visually by Its Similarity Matrix

The previous technique suggests a more general, qualitative approach to judging a set of clusters: Order the similarity matrix with respect to cluster labels and then plot it. In theory, if we have well-separated clusters, then the similarity matrix should be roughly block-diagonal. If not, then the patterns displayed in the similarity matrix can reveal the relationships between clusters. Again, all of this can be applied to dissimilarity matrices, but for simplicity, we will only discuss similarity matrices.

Example 8.10 (Visualizing a Similarity Matrix). Consider the points in Figure 8.30(a), which form three well-separated clusters. If we use K-means to group these points into three clusters, then we should have no trouble finding these clusters since they are well-separated. The separation of these clusters is illustrated by the reordered similarity matrix shown in Figure 8.30(b). (For uniformity, we have transformed the distances into similarities using the formula $s = 1 - (d - min\_d)/(max\_d - min\_d)$.) Figure 8.31 shows the reordered similarity matrices for clusters found in the random data set of Figure 8.26 by DBSCAN, K-means, and complete link.



(a) Well-separated clusters.

(b) Similarity matrix sorted by K-means cluster labels.

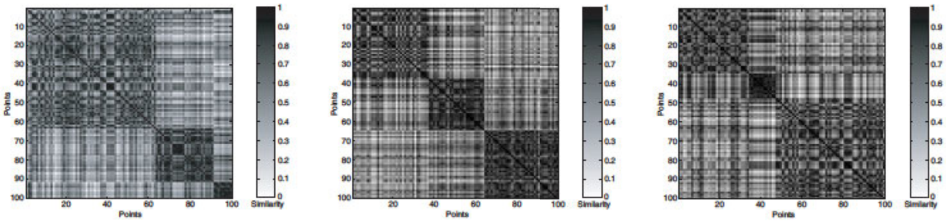
Figure 8.30. Similarity matrix for well-separated clusters.

The well-separated clusters in Figure 8.30 show a very strong, block-diagonal pattern in the reordered similarity matrix. However, there are also weak block diagonal patterns—see Figure 8.31—in the reordered similarity matrices of the clusterings found by K-means, DBSCAN, and complete link in the random data. Just as people can find patterns in clouds, data mining algorithms can find clusters in random data. While it is entertaining to find patterns in clouds, it is pointless and perhaps embarrassing to find clusters in noise. ■

This approach may seem hopelessly expensive for large data sets, since the computation of the proximity matrix takes $O(m^2)$ time, where m is the number of objects, but with sampling, this method can still be used. We can take a sample of data points from each cluster, compute the similarity between these points, and plot the result. It may be necessary to oversample small clusters and undersample large ones to obtain an adequate representation of all clusters.

8.5.4 Unsupervised Evaluation of Hierarchical Clustering

The previous approaches to cluster evaluation are intended for partitional clusterings. Here we discuss the cophenetic correlation, a popular evaluation measure for hierarchical clusterings. The **cophenetic distance** between two objects is the proximity at which an agglomerative hierarchical clustering tech-



(a) Similarity matrix sorted by DBSCAN cluster labels.

(b) Similarity matrix sorted by K-means cluster labels.

(c) Similarity matrix sorted by complete link cluster labels.

Figure 8.31. Similarity matrices for clusters from random data.

nique puts the objects in the same cluster for the first time. For example, if at some point in the agglomerative hierarchical clustering process, the smallest distance between the two clusters that are merged is 0.1, then all points in one cluster have a cophenetic distance of 0.1 with respect to the points in the other cluster. In a cophenetic distance matrix, the entries are the cophenetic distances between each pair of objects. The cophenetic distance is different for each hierarchical clustering of a set of points.

Example 8.11 (Cophenetic Distance Matrix). Table 8.7 shows the cophentic distance matrix for the single link clustering shown in Figure 8.16. (The data for this figure consists of the 6 two-dimensional points given in Table 8.3.)

Table 8.7. Cophenetic distance matrix for single link and data in table 8.3

Point	P1	P2	P3	P4	P5	P6
P1	0	0.222	0.222	0.222	0.222	0.222
P2	0.222	0	0.148	0.151	0.139	0.148
P3	0.222	0.148	0	0.151	0.148	0.110
P4	0.222	0.151	0.151	0	0.151	0.151
P5	0.222	0.139	0.148	0.151	0	0.148
P6	0.222	0.148	0.110	0.151	0.148	0

■

The **CoPhenetic Correlation Coefficient** (CPCC) is the correlation between the entries of this matrix and the original dissimilarity matrix and is a standard measure of how well a hierarchical clustering (of a particular type) fits the data. One of the most common uses of this measure is to evaluate which type of hierarchical clustering is best for a particular type of data.

Example 8.12 (Cophenetic Correlation Coefficient). We calculated the CPCC for the hierarchical clusterings shown in Figures 8.16–8.19.These values are shown in Table 8.8. The hierarchical clustering produced by the single link technique seems to fit the data less well than the clusterings produced by complete link, group average, and Ward's method.

Table 8.8. Cophenetic correlation coefficient for data of Table 8.3 and four agglomerative hierarchical clustering techniques.

Technique	CPCC
Single Link	0.44
Complete Link	0.63
Group Average	0.66
Ward's	0.64

■

8.5.5 Determining the Correct Number of Clusters

Various unsupervised cluster evaluation measures can be used to approximately determine the correct or natural number of clusters.

Example 8.13 (Number of Clusters). The data set of Figure 8.29 has 10 natural clusters. Figure 8.32 shows a plot of the SSE versus the number of clusters for a (bisecting) K-means clustering of the data set, while Figure 8.33 shows the average silhouette coefficient versus the number of clusters for the same data. There is a distinct knee in the SSE and a distinct peak in the silhouette coefficient when the number of clusters is equal to 10. ■

Thus, we can try to find the natural number of clusters in a data set by looking for the number of clusters at which there is a knee, peak, or dip in the plot of the evaluation measure when it is plotted against the number of clusters. Of course, such an approach does not always work well. Clusters may be considerably more intertwined or overlapping than those shown in Figure 8.29. Also, the data may consist of nested clusters. Actually, the clusters in Figure 8.29 are somewhat nested; i.e., there are 5 pairs of clusters since the clusters are closer top to bottom than they are left to right. There is a knee that indicates this in the SSE curve, but the silhouette coefficient curve is not

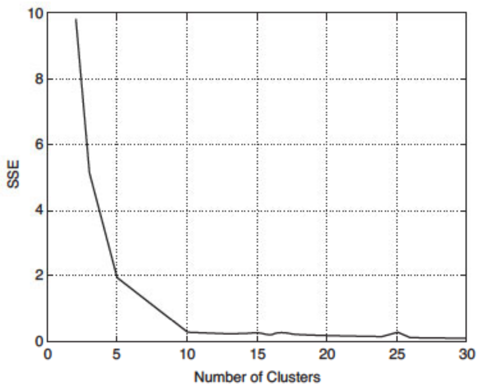


Figure 8.32. SSE versus number of clusters for the data of Figure 8.29.

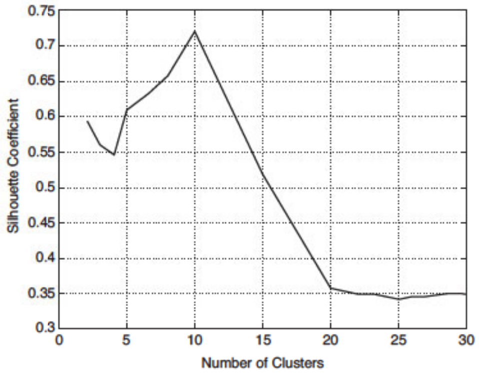


Figure 8.33. Average silhouette coefficient versus number of clusters for the data of Figure 8.29.

as clear. In summary, while caution is needed, the technique we have just described can provide insight into the number of clusters in the data.

8.5.6 Clustering Tendency

One obvious way to determine if a data set has clusters is to try to cluster it. However, almost all clustering algorithms will dutifully find clusters when given data. To address this issue, we could evaluate the resulting clusters and only claim that a data set has clusters if at least some of the clusters are of good quality. However, this approach does not address the fact the clusters in the data can be of a different type than those sought by our clustering algorithm. To handle this additional problem, we could use multiple algorithms and again evaluate the quality of the resulting clusters. If the clusters are uniformly poor, then this may indeed indicate that there are no clusters in the data.

Alternatively, and this is the focus of measures of clustering tendency, we can try to evaluate whether a data set has clusters without clustering. The most common approach, especially for data in Euclidean space, has been to use statistical tests for spatial randomness. Unfortunately, choosing the correct model, estimating the parameters, and evaluating the statistical significance of the hypothesis that the data is non-random can be quite challenging. Nonetheless, many approaches have been developed, most of them for points in low-dimensional Euclidean space.

Example 8.14 (Hopkins Statistic). For this approach, we generate p points that are randomly distributed across the data space and also sample p actual data points. For both sets of points we find the distance to the nearest neighbor in the original data set. Let the u_i be the nearest neighbor distances of the artificially generated points, while the w_i are the nearest neighbor distances of the sample of points from the original data set. The Hopkins statistic H is then defined by Equation 8.17.

$$H = \frac{\sum_{i=1}^{p} w_i}{\sum_{i=1}^{p} u_i + \sum_{i=1}^{p} w_i} \tag{8.17}$$

If the randomly generated points and the sample of data points have roughly the same nearest neighbor distances, then H will be near 0.5. Values of H near 0 and 1 indicate, respectively, data that is highly clustered and data that is regularly distributed in the data space. To give an example, the Hopkins statistic for the data of Figure 8.26 was computed for $p = 20$ and 100 different trials. The average value of H was 0.56 with a standard deviation of 0.03. The same experiment was performed for the well-separated points of Figure 8.30. The average value of H was 0.95 with a standard deviation of 0.006. ■

8.5.7 Supervised Measures of Cluster Validity

When we have external information about data, it is typically in the form of externally derived class labels for the data objects. In such cases, the usual procedure is to measure the degree of correspondence between the cluster labels and the class labels. But why is this of interest? After all, if we have the class labels, then what is the point in performing a cluster analysis? Motivations for such an analysis are the comparison of clustering techniques with the "ground truth" or the evaluation of the extent to which a manual classification process can be automatically produced by cluster analysis.

We consider two different kinds of approaches. The first set of techniques use measures from classification, such as entropy, purity, and the F-measure. These measures evaluate the extent to which a cluster contains objects of a single class. The second group of methods is related to the similarity measures for binary data, such as the Jaccard measure that we saw in Chapter 2. These approaches measure the extent to which two objects that are in the same class are in the same cluster and vice versa. For convenience, we will refer to these two types of measures as **classification-oriented** and **similarity-oriented**, respectively.

Classification-Oriented Measures of Cluster Validity

There are a number of measures—entropy, purity, precision, recall, and the F-measure—that are commonly used to evaluate the performance of a classification model. In the case of classification, we measure the degree to which predicted class labels correspond to actual class labels, but for the measures just mentioned, nothing fundamental is changed by using cluster labels instead of predicted class labels. Next, we quickly review the definitions of these measures, which were discussed in Chapter 4.

Entropy: The degree to which each cluster consists of objects of a single class. For each cluster, the class distribution of the data is calculated first, i.e., for cluster j we compute p_{ij}, the probability that a member of cluster i belongs to class j as $p_{ij} = m_{ij}/m_i$, where m_i is the number of objects in cluster i and m_{ij} is the number of objects of class j in cluster i. Using this class distribution, the entropy of each cluster i is calculated using the standard formula, $e_i = -\sum_{j=1}^{L} p_{ij} \log_2 p_{ij}$, where L is the number of classes. The total entropy for a set of clusters is calculated as the sum of the entropies of each cluster weighted by the size of each cluster, i.e., $e = \sum_{i=1}^{K} \frac{m_i}{m} e_i$, where K is the number of clusters and m is the total number of data points.

Purity: Another measure of the extent to which a cluster contains objects of a single class. Using the previous terminology, the purity of cluster i is $p_i = \max_{j} p_{ij}$, the overall purity of a clustering is $purity = \sum_{i=1}^{K} \frac{m_i}{m} p_i$.

Precision: The fraction of a cluster that consists of objects of a specified class. The precision of cluster i with respect to class j is $precision(i, j) = p_{ij}$.

Recall: The extent to which a cluster contains all objects of a specified class. The recall of cluster i with respect to class j is $recall(i, j) = m_{ij}/m_j$, where m_j is the number of objects in class j.

F-measure A combination of both precision and recall that measures the extent to which a cluster contains *only* objects of a particular class and *all* objects of that class. The F-measure of cluster i with respect to class j is $F(i, j) = (2 \times precision(i, j) \times recall(i, j))/(precision(i, j) + recall(i, j))$.

Example 8.15 (Supervised Evaluation Measures). We present an example to illustrate these measures. Specifically, we use K-means with the cosine similarity measure to cluster 3204 newspaper articles from the *Los Angeles*

Table 8.9. K-means clustering results for the *LA Times* document data set.

Cluster	Entertainment	Financial	Foreign	Metro	National	Sports	Entropy	Purity
1	3	5	40	506	96	27	1.2270	0.7474
2	4	7	280	29	39	2	1.1472	0.7756
3	1	1	1	7	4	671	0.1813	0.9796
4	10	162	3	119	73	2	1.7487	0.4390
5	331	22	5	70	13	23	1.3976	0.7134
6	5	358	12	212	48	13	1.5523	0.5525
Total	354	555	341	943	273	738	1.1450	0.7203

Times. These articles come from six different classes: Entertainment, Financial, Foreign, Metro, National, and Sports. Table 8.9 shows the results of a K-means clustering to find six clusters. The first column indicates the cluster, while the next six columns together form the confusion matrix; i.e., these columns indicate how the documents of each category are distributed among the clusters. The last two columns are the entropy and purity of each cluster, respectively.

Ideally, each cluster will contain documents from only one class. In reality, each cluster contains documents from many classes. Nevertheless, many clusters contain documents primarily from just one class. In particular, cluster 3, which contains mostly documents from the Sports section, is exceptionally good, both in terms of purity and entropy. The purity and entropy of the other clusters is not as good, but can typically be greatly improved if the data is partitioned into a larger number of clusters.

Precision, recall, and the F-measure can be calculated for each cluster. To give a concrete example, we consider cluster 1 and the Metro class of Table 8.9. The precision is $506/677 = 0.75$, recall is $506/943 = 0.26$, and hence, the F value is 0.39. In contrast, the F value for cluster 3 and Sports is 0.94. ■

Similarity-Oriented Measures of Cluster Validity

The measures that we discuss in this section are all based on the premise that any two objects that are in the same cluster should be in the same class and vice versa. We can view this approach to cluster validity as involving the comparison of two matrices: (1) the **ideal cluster similarity matrix** discussed previously, which has a 1 in the ij^{th} entry if two objects, i and j, are in the same cluster and 0, otherwise, and (2) an **ideal class similarity matrix** defined with respect to class labels, which has a 1 in the ij^{th} entry if two objects, i and j, belong to the same class, and a 0 otherwise. As before, we can take the correlation of these two matrices as the measure of cluster validity. This measure is known as the Γ statistic in clustering validation literature.

Example 8.16 (Correlation between Cluster and Class Matrices). To demonstrate this idea more concretely, we give an example involving five data points, p_1, p_2, p_3, p_4, p_5, two clusters, $C_1 = \{p_1, p_2, p_3\}$ and $C_2 = \{p_4, p_5\}$, and two classes, $L_1 = \{p_1, p_2\}$ and $L2 = \{p_3, p_4, p_5\}$. The ideal cluster and class similarity matrices are given in Tables 8.10 and 8.11. The correlation between the entries of these two matrices is 0.359.

Table 8.10. Ideal cluster similarity matrix.

Point	p1	p2	p3	p4	p5
p1	1	1	1	0	0
p2	1	1	1	0	0
p3	1	1	1	0	0
p4	0	0	0	1	1
p5	0	0	0	1	1

Table 8.11. Ideal class similarity matrix.

Point	p1	p2	p3	p4	p5
p1	1	1	0	0	0
p2	1	1	0	0	0
p3	0	0	1	1	1
p4	0	0	1	1	1
p5	0	0	1	1	1

■

More generally, we can use any of the measures for binary similarity that we saw in Section 2.4.5. (For example, we can convert these two matrices into binary vectors by appending the rows.) We repeat the definitions of the four quantities used to define those similarity measures, but modify our descriptive text to fit the current context. Specifically, we need to compute the following four quantities for all pairs of distinct objects. (There are $m(m-1)/2$ such pairs, if m is the number of objects.)

f_{00} = number of pairs of objects having a different class and a different cluster
f_{01} = number of pairs of objects having a different class and the same cluster
f_{10} = number of pairs of objects having the same class and a different cluster
f_{11} = number of pairs of objects having the same class and the same cluster

In particular, the simple matching coefficient, which is known as the Rand statistic in this context, and the Jaccard coefficient are two of the most frequently used cluster validity measures.

$$\text{Rand statistic} = \frac{f_{00} + f_{11}}{f_{00} + f_{01} + f_{10} + f_{11}} \tag{8.18}$$

$$\text{Jaccard coefficient} = \frac{f_{11}}{f_{01} + f_{10} + f_{11}} \tag{8.19}$$

Example 8.17 (Rand and Jaccard Measures). Based on these formulas, we can readily compute the Rand statistic and Jaccard coefficient for the example based on Tables 8.10 and 8.11. Noting that $f_{00} = 4$, $f_{01} = 2$, $f_{10} = 2$, and $f_{11} = 2$, the Rand statistic $= (2 + 4)/10 = 0.6$ and the Jaccard coefficient $= 2/(2{+}2{+}2\ ) = 0.33$. ■

We also note that the four quantities, f_{00}, f_{01}, f_{10}, and f_{11}, define a *contingency* table as shown in Table 8.12.

Table 8.12. Two-way contingency table for determining whether pairs of objects are in the same class and same cluster.

	Same Cluster	Different Cluster
Same Class	f_{11}	f_{10}
Different Class	f_{01}	f_{00}

Previously, in the context of association analysis—see Section 6.7.1—we presented an extensive discussion of measures of association that can be used for this type of contingency table. (Compare Table 8.12 with Table 6.7.) Those measures can also be applied to cluster validity.

Cluster Validity for Hierarchical Clusterings

So far in this section, we have discussed supervised measures of cluster validity only for partitional clusterings. Supervised evaluation of a hierarchical clustering is more difficult for a variety of reasons, including the fact that a preexisting hierarchical structure often does not exist. Here, we will give an example of an approach for evaluating a hierarchical clustering in terms of a (flat) set of class labels, which are more likely to be available than a preexisting hierarchical structure.

The key idea of this approach is to evaluate whether a hierarchical clustering contains, for each class, at least one cluster that is relatively pure and includes most of the objects of that class. To evaluate a hierarchical clustering with respect to this goal, we compute, for each class, the F-measure for each cluster in the cluster hierarchy. For each class, we take the maximum F-measure attained for any cluster. Finally, we calculate an overall F-measure for the hierarchical clustering by computing the weighted average of all per-class F-measures, where the weights are based on the class sizes. More formally, this hierarchical F-measure is defined as follows:

$$F = \sum_j \frac{m_j}{m} \max_i F(i,j)$$

where the maximum is taken over all clusters i at all levels, m_j is the number of objects in class j, and m is the total number of objects.

8.5.8 Assessing the Significance of Cluster Validity Measures

Cluster validity measures are intended to help us measure the goodness of the clusters that we have obtained. Indeed, they typically give us a single number as a measure of that goodness. However, we are then faced with the problem of interpreting the significance of this number, a task that may be even more difficult.

The minimum and maximum values of cluster evaluation measures may provide some guidance in many cases. For instance, by definition, a purity of 0 is bad, while a purity of 1 is good, at least if we trust our class labels and want our cluster structure to reflect the class structure. Likewise, an entropy of 0 is good, as is an SSE of 0.

Sometimes, however, there may not be a minimum or maximum value, or the scale of the data may affect the interpretation. Also, even if there are minimum and maximum values with obvious interpretations, intermediate values still need to be interpreted. In some cases, we can use an absolute standard. If, for example, we are clustering for utility, we may be willing to tolerate only a certain level of error in the approximation of our points by a cluster centroid.

But if this is not the case, then we must do something else. A common approach is to interpret the value of our validity measure in statistical terms. Specifically, we attempt to judge how likely it is that our observed value may be achieved by random chance. The value is good if it is unusual; i.e., if it is unlikely to be the result of random chance. The motivation for this approach is that we are only interested in clusters that reflect non-random structure in the data, and such structures should generate unusually high (low) values of our cluster validity measure, at least if the validity measures are designed to reflect the presence of strong cluster structure.

Example 8.18 (Significance of SSE). To show how this works, we present an example based on K-means and the SSE. Suppose that we want a measure of how good the well-separated clusters of Figure 8.30 are with respect to random data. We generate many random sets of 100 points having the same range as

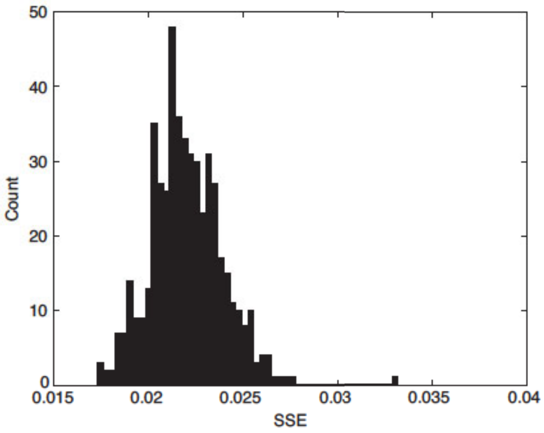


Figure 8.34. Histogram of SSE for 500 random data sets.

the points in the three clusters, find three clusters in each data set using K-means, and accumulate the distribution of SSE values for these clusterings. By using this distribution of the SSE values, we can then estimate the probability of the SSE value for the original clusters. Figure 8.34 shows the histogram of the SSE from 500 random runs. The lowest SSE shown in Figure 8.34 is 0.0173. For the three clusters of Figure 8.30, the SSE is 0.0050. We could therefore conservatively claim that there is less than a 1% chance that a clustering such as that of Figure 8.30 could occur by chance. ■

To conclude, we stress that there is more to cluster evaluation—supervised or unsupervised—than obtaining a numerical measure of cluster validity. Unless this value has a natural interpretation based on the definition of the measure, we need to interpret this value in some way. If our cluster evaluation measure is defined such that lower values indicate stronger clusters, then we can use statistics to evaluate whether the value we have obtained is unusually low, provided we have a distribution for the evaluation measure. We have presented an example of how to find such a distribution, but there is considerably more to this topic, and we refer the reader to the bibliographic notes for more pointers.

Finally, even when an evaluation measure is used as a relative measure, i.e., to compare two clusterings, we still need to assess the significance in the difference between the evaluation measures of the two clusterings. Although one value will almost always be better than another, it can be difficult to determine if the difference is significant. Note that there are two aspects to this significance: whether the difference is statistically significant (repeatable) and whether the magnitude of the difference is meaningful with respect to the application. Many would not regard a difference of 0.1% as significant, even if it is consistently reproducible.

8.6 Bibliographic Notes

Discussion in this chapter has been most heavily influenced by the books on cluster analysis written by Jain and Dubes [396], Anderberg [374], and Kaufman and Rousseeuw [400]. Additional clustering books that may also be of interest include those by Aldenderfer and Blashfield [373], Everitt et al. [388], Hartigan [394], Mirkin [405], Murtagh [407], Romesburg [409], and Späth [413]. A more statistically oriented approach to clustering is given by the pattern recognition book of Duda et al. [385], the machine learning book of Mitchell [406], and the book on statistical learning by Hastie et al. [395]. A general survey of clustering is given by Jain et al. [397], while a survey of spatial data mining techniques is provided by Han et al. [393]. Behrkin [379] provides a survey of clustering techniques for data mining. A good source of references to clustering outside of the data mining field is the article by Arabie and Hubert [376]. A paper by Kleinberg [401] provides a discussion of some of the trade-offs that clustering algorithms make and proves that it is impossible to for a clustering algorithm to simultaneously possess three simple properties.

The K-means algorithm has a long history, but is still the subject of current research. The original K-means algorithm was proposed by MacQueen [403]. The ISODATA algorithm by Ball and Hall [377] was an early, but sophisticated version of K-means that employed various pre- and postprocessing techniques to improve on the basic algorithm. The K-means algorithm and many of its variations are described in detail in the books by Anderberg [374] and Jain and Dubes [396]. The bisecting K-means algorithm discussed in this chapter was described in a paper by Steinbach et al. [414], and an implementation of this and other clustering approaches is freely available for academic use in the CLUTO (CLUstering TOolkit) package created by Karypis [382]. Boley [380] has created a divisive partitioning clustering algorithm (PDDP) based on finding the first principal direction (component) of the data, and Savaresi and Boley [411] have explored its relationship to bisecting K-means. Recent variations of K-means are a new incremental version of K-means (Dhillon et al. [383]), X-means (Pelleg and Moore [408]), and K-harmonic means (Zhang et al [416]). Hamerly and Elkan [392] discuss some clustering algorithms that produce better results than K-means. While some of the previously mentioned approaches address the initialization problem of K-means in some manner, other approaches to improving K-means initialization can also be found in the work of Bradley and Fayyad [381]. Dhillon and Modha [384] present a generalization of K-means, called spherical K-means, that works with commonly used similarity functions. A general framework for K-means clustering that uses dissimilarity functions based on Bregman divergences was constructed by Banerjee et al. [378].

Hierarchical clustering techniques also have a long history. Much of the initial activity was in the area of taxonomy and is covered in books by Jardine and Sibson [398] and Sneath and Sokal [412]. General-purpose discussions of hierarchical clustering are also available in most of the clustering books mentioned above. Agglomerative hierarchical clustering is the focus of most work in the area of hierarchical clustering, but divisive approaches have also received some attention. For example, Zahn [415] describes a divisive hierarchical technique that uses the minimum spanning tree of a graph. While both divisive and agglomerative approaches typically take the view that merging (splitting) decisions are final, there has been some work by Fisher [389] and Karypis et al. [399] to overcome these limitations.

Ester et al. proposed DBSCAN [387], which was later generalized to the GDBSCAN algorithm by Sander et al. [410] in order to handle more general types of data and distance measures, such as polygons whose closeness is measured by the degree of intersection. An incremental version of DBSCAN was developed by Kriegel et al. [386]. One interesting outgrowth of DBSCAN is OPTICS (Ordering Points To Identify the Clustering Structure) (Ankerst et al. [375]), which allows the visualization of cluster structure and can also be used for hierarchical clustering.

An authoritative discussion of cluster validity, which strongly influenced the discussion in this chapter, is provided in Chapter 4 of Jain and Dubes' clustering book [396]. More recent reviews of cluster validity are those of Halkidi et al. [390, 391] and Milligan [404]. Silhouette coefficients are described in Kaufman and Rousseeuw's clustering book [400]. The source of the cohesion and separation measures in Table 8.6 is a paper by Zhao and Karypis [417], which also contains a discussion of entropy, purity, and the hierarchical F-measure. The original source of the hierarchical F-measure is an article by Larsen and Aone [402].

Bibliography

[373] M. S. Aldenderfer and R. K. Blashfield. *Cluster Analysis.* Sage Publications, Los Angeles, 1985.

[374] M. R. Anderberg. *Cluster Analysis for Applications.* Academic Press, New York, December 1973.

5

Classification: Alternative Techniques

The previous chapter described a simple, yet quite effective, classification technique known as decision tree induction. Issues such as model overfitting and classifier evaluation were also discussed in great detail. This chapter presents alternative techniques for building classification models—from simple techniques such as rule-based and nearest-neighbor classifiers to more advanced techniques such as support vector machines and ensemble methods. Other key issues such as the class imbalance and multiclass problems are also discussed at the end of the chapter.

5.1 Rule-Based Classifier

A rule-based classifier is a technique for classifying records using a collection of "if ...then..." rules. Table 5.1 shows an example of a model generated by a rule-based classifier for the vertebrate classification problem. The rules for the model are represented in a disjunctive normal form, $R = (r_1 \vee r_2 \vee \ldots r_k)$, where R is known as the **rule set** and r_i's are the classification rules or disjuncts.

Table 5.1. Example of a rule set for the vertebrate classification problem.

r_1:	(Gives Birth = no) $\wedge$ (Aerial Creature = yes) $\longrightarrow$ Birds
r_2:	(Gives Birth = no) $\wedge$ (Aquatic Creature = yes) $\longrightarrow$ Fishes
r_3:	(Gives Birth = yes) $\wedge$ (Body Temperature = warm-blooded) $\longrightarrow$ Mammals
r_4:	(Gives Birth = no) $\wedge$ (Aerial Creature = no) $\longrightarrow$ Reptiles
r_5:	(Aquatic Creature = semi) $\longrightarrow$ Amphibians

Each classification rule can be expressed in the following way:

$$r_i: \quad (Condition_i) \longrightarrow y_i. \tag{5.1}$$

The left-hand side of the rule is called the **rule antecedent** or **precondition**. It contains a conjunction of attribute tests:

$$Condition_i = (A_1 \; op \; v_1) \wedge (A_2 \; op \; v_2) \wedge \ldots (A_k \; op \; v_k), \tag{5.2}$$

where (A_j, v_j) is an attribute-value pair and *op* is a logical operator chosen from the set $\{=, \neq, <, >, \leq, \geq\}$. Each attribute test $(A_j \; op \; v_j)$ is known as a conjunct. The right-hand side of the rule is called the **rule consequent**, which contains the predicted class y_i.

A rule r covers a record x if the precondition of r matches the attributes of x. r is also said to be fired or triggered whenever it covers a given record. For an illustration, consider the rule r_1 given in Table 5.1 and the following attributes for two vertebrates: hawk and grizzly bear.

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hiber-nates
hawk	warm-blooded	feather	no	no	yes	yes	no
grizzly bear	warm-blooded	fur	yes	no	no	yes	yes

r_1 covers the first vertebrate because its precondition is satisfied by the hawk's attributes. The rule does not cover the second vertebrate because grizzly bears give birth to their young and cannot fly, thus violating the precondition of r_1.

The quality of a classification rule can be evaluated using measures such as coverage and accuracy. Given a data set D and a classification rule $r : A \longrightarrow y$, the coverage of the rule is defined as the fraction of records in D that trigger the rule r. On the other hand, its accuracy or confidence factor is defined as the fraction of records triggered by r whose class labels are equal to y. The formal definitions of these measures are

$$\begin{aligned} \text{Coverage}(r) &= \frac{|A|}{|D|} \\ \text{Accuracy}(r) &= \frac{|A \cap y|}{|A|}, \end{aligned} \tag{5.3}$$

where $|A|$ is the number of records that satisfy the rule antecedent, $|A \cap y|$ is the number of records that satisfy both the antecedent and consequent, and $|D|$ is the total number of records.

Table 5.2. The vertebrate data set.

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hiber-nates	Class Label
human	warm-blooded	hair	yes	no	no	yes	no	Mammals
python	cold-blooded	scales	no	no	no	no	yes	Reptiles
salmon	cold-blooded	scales	no	yes	no	no	no	Fishes
whale	warm-blooded	hair	yes	yes	no	no	no	Mammals
frog	cold-blooded	none	no	semi	no	yes	yes	Amphibians
komodo dragon	cold-blooded	scales	no	no	no	yes	no	Reptiles
bat	warm-blooded	hair	yes	no	yes	yes	yes	Mammals
pigeon	warm-blooded	feathers	no	no	yes	yes	no	Birds
cat	warm-blooded	fur	yes	no	no	yes	no	Mammals
guppy	cold-blooded	scales	yes	yes	no	no	no	Fishes
alligator	cold-blooded	scales	no	semi	no	yes	no	Reptiles
penguin	warm-blooded	feathers	no	semi	no	yes	no	Birds
porcupine	warm-blooded	quills	yes	no	no	yes	yes	Mammals
eel	cold-blooded	scales	no	yes	no	no	no	Fishes
salamander	cold-blooded	none	no	semi	no	yes	yes	Amphibians

Example 5.1. Consider the data set shown in Table 5.2. The rule

$$(\texttt{Gives Birth} = \texttt{yes}) \wedge (\texttt{Body Temperature} = \texttt{warm-blooded}) \longrightarrow \texttt{Mammals}$$

has a coverage of 33% since five of the fifteen records support the rule antecedent. The rule accuracy is 100% because all five vertebrates covered by the rule are mammals. ■

5.1.1 How a Rule-Based Classifier Works

A rule-based classifier classifies a test record based on the rule triggered by the record. To illustrate how a rule-based classifier works, consider the rule set shown in Table 5.1 and the following vertebrates:

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hiber-nates
lemur	warm-blooded	fur	yes	no	no	yes	yes
turtle	cold-blooded	scales	no	semi	no	yes	no
dogfish shark	cold-blooded	scales	yes	yes	no	no	no

- The first vertebrate, which is a lemur, is warm-blooded and gives birth to its young. It triggers the rule r_3, and thus, is classified as a mammal.

- The second vertebrate, which is a turtle, triggers the rules r_4 and r_5. Since the classes predicted by the rules are contradictory (reptiles versus amphibians), their conflicting classes must be resolved.

- None of the rules are applicable to a dogfish shark. In this case, we need to ensure that the classifier can still make a reliable prediction even though a test record is not covered by any rule.

The previous example illustrates two important properties of the rule set generated by a rule-based classifier.

Mutually Exclusive Rules The rules in a rule set R are mutually exclusive if no two rules in R are triggered by the same record. This property ensures that every record is covered by at most one rule in R. An example of a mutually exclusive rule set is shown in Table 5.3.

Exhaustive Rules A rule set R has exhaustive coverage if there is a rule for each combination of attribute values. This property ensures that every record is covered by at least one rule in R. Assuming that `Body Temperature` and `Gives Birth` are binary variables, the rule set shown in Table 5.3 has exhaustive coverage.

Table 5.3. Example of a mutually exclusive and exhaustive rule set.

r_1: (Body Temperature = cold-blooded) $\longrightarrow$ Non-mammals
r_2: (Body Temperature = warm-blooded) $\wedge$ (Gives Birth = yes) $\longrightarrow$ Mammals
r_3: (Body Temperature = warm-blooded) $\wedge$ (Gives Birth = no) $\longrightarrow$ Non-mammals

Together, these properties ensure that every record is covered by exactly one rule. Unfortunately, many rule-based classifiers, including the one shown in Table 5.1, do not have such properties. If the rule set is not exhaustive, then a default rule, r_d : $() \longrightarrow y_d$, must be added to cover the remaining cases. A default rule has an empty antecedent and is triggered when all other rules have failed. y_d is known as the default class and is typically assigned to the majority class of training records not covered by the existing rules.

If the rule set is not mutually exclusive, then a record can be covered by several rules, some of which may predict conflicting classes. There are two ways to overcome this problem.

Ordered Rules In this approach, the rules in a rule set are ordered in decreasing order of their priority, which can be defined in many ways (e.g., based on accuracy, coverage, total description length, or the order in which the rules are generated). An ordered rule set is also known as a **decision list**. When a test record is presented, it is classified by the highest-ranked rule that covers the record. This avoids the problem of having conflicting classes predicted by multiple classification rules.

Unordered Rules This approach allows a test record to trigger multiple classification rules and considers the consequent of each rule as a vote for a particular class. The votes are then tallied to determine the class label of the test record. The record is usually assigned to the class that receives the highest number of votes. In some cases, the vote may be weighted by the rule's accuracy. Using unordered rules to build a rule-based classifier has both advantages and disadvantages. Unordered rules are less susceptible to errors caused by the wrong rule being selected to classify a test record (unlike classifiers based on ordered rules, which are sensitive to the choice of rule-ordering criteria). Model building is also less expensive because the rules do not have to be kept in sorted order. Nevertheless, classifying a test record can be quite an expensive task because the attributes of the test record must be compared against the precondition of every rule in the rule set.

In the remainder of this section, we will focus on rule-based classifiers that use ordered rules.

5.1.2 Rule-Ordering Schemes

Rule ordering can be implemented on a rule-by-rule basis or on a class-by-class basis. The difference between these schemes is illustrated in Figure 5.1.

Rule-Based Ordering Scheme This approach orders the individual rules by some rule quality measure. This ordering scheme ensures that every test record is classified by the "best" rule covering it. A potential drawback of this scheme is that lower-ranked rules are much harder to interpret because they assume the negation of the rules preceding them. For example, the fourth rule shown in Figure 5.1 for rule-based ordering,

$$\text{Aquatic Creature} = \text{semi} \longrightarrow \text{Amphibians},$$

has the following interpretation: If the vertebrate does not have any feathers or cannot fly, and is cold-blooded and semi-aquatic, then it is an amphibian.

Rule-Based Ordering

(Skin Cover=feathers, Aerial Creature=yes)
==> Birds

(Body temperature=warm-blooded,
Gives Birth=yes) ==> Mammals

(Body temperature=warm-blooded,
Gives Birth=no) ==> Birds

(Aquatic Creature=semi)) ==> Amphibians

(Skin Cover=scales, Aquatic Creature=no)
==> Reptiles

(Skin Cover=scales, Aquatic Creature=yes)
==> Fishes

(Skin Cover=none) ==> Amphibians

Class-Based Ordering

(Skin Cover=feathers, Aerial Creature=yes)
==> Birds

(Body temperature=warm-blooded,
Gives Birth=no) ==> Birds

(Body temperature=warm-blooded,
Gives Birth=yes) ==> Mammals

(Aquatic Creature=semi)) ==> Amphibians

(Skin Cover=none) ==> Amphibians

(Skin Cover=scales, Aquatic Creature=no)
==> Reptiles

(Skin Cover=scales, Aquatic Creature=yes)
==> Fishes

Figure 5.1. Comparison between rule-based and class-based ordering schemes.

The additional conditions (that the vertebrate does not have any feathers or cannot fly, and is cold-blooded) are due to the fact that the vertebrate does not satisfy the first three rules. If the number of rules is large, interpreting the meaning of the rules residing near the bottom of the list can be a cumbersome task.

Class-Based Ordering Scheme In this approach, rules that belong to the same class appear together in the rule set R. The rules are then collectively sorted on the basis of their class information. The relative ordering among the rules from the same class is not important; as long as one of the rules fires, the class will be assigned to the test record. This makes rule interpretation slightly easier. However, it is possible for a high-quality rule to be overlooked in favor of an inferior rule that happens to predict the higher-ranked class.

Since most of the well-known rule-based classifiers (such as C4.5rules and RIPPER) employ the class-based ordering scheme, the discussion in the remainder of this section focuses mainly on this type of ordering scheme.

5.1.3 How to Build a Rule-Based Classifier

To build a rule-based classifier, we need to extract a set of rules that identifies key relationships between the attributes of a data set and the class label.

There are two broad classes of methods for extracting classification rules: (1) direct methods, which extract classification rules directly from data, and (2) indirect methods, which extract classification rules from other classification models, such as decision trees and neural networks.

Direct methods partition the attribute space into smaller subspaces so that all the records that belong to a subspace can be classified using a single classification rule. Indirect methods use the classification rules to provide a succinct description of more complex classification models. Detailed discussions of these methods are presented in Sections 5.1.4 and 5.1.5, respectively.

5.1.4 Direct Methods for Rule Extraction

The **sequential covering** algorithm is often used to extract rules directly from data. Rules are grown in a greedy fashion based on a certain evaluation measure. The algorithm extracts the rules one class at a time for data sets that contain more than two classes. For the vertebrate classification problem, the sequential covering algorithm may generate rules for classifying birds first, followed by rules for classifying mammals, amphibians, reptiles, and finally, fishes (see Figure 5.1). The criterion for deciding which class should be generated first depends on a number of factors, such as the class prevalence (i.e., fraction of training records that belong to a particular class) or the cost of misclassifying records from a given class.

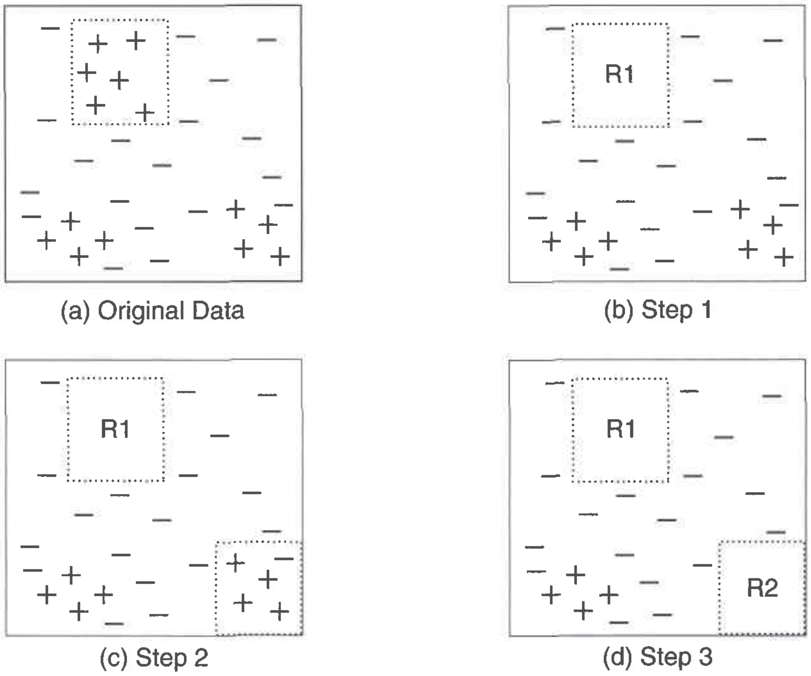
A summary of the sequential covering algorithm is given in Algorithm 5.1. The algorithm starts with an empty decision list, R. The Learn-One-Rule function is then used to extract the best rule for class y that covers the current set of training records. During rule extraction, all training records for class y are considered to be positive examples, while those that belong to

Algorithm 5.1 Sequential covering algorithm.

```
Let $E$ be the training records and $A$ be the set of attribute-value pairs, $\{(A_j, v_j)\}$.
Let $Y_o$ be an ordered set of classes $\{y_1, y_2, \ldots, y_k\}$.
Let $R = \{\ \}$ be the initial rule list.
for each class $y \in Y_o - \{y_k\}$ do
  while stopping condition is not met do
    $r \leftarrow$ Learn-One-Rule ($E$, $A$, $y$).
    Remove training records from $E$ that are covered by $r$.
    Add $r$ to the bottom of the rule list: $R \longrightarrow R \vee r$.
  end while
end for
Insert the default rule, $\{\} \longrightarrow y_k$, to the bottom of the rule list $R$.
```

other classes are considered to be negative examples. A rule is desirable if it covers most of the positive examples and none (or very few) of the negative examples. Once such a rule is found, the training records covered by the rule are eliminated. The new rule is added to the bottom of the decision list R. This procedure is repeated until the stopping criterion is met. The algorithm then proceeds to generate rules for the next class.

Figure 5.2 demonstrates how the sequential covering algorithm works for a data set that contains a collection of positive and negative examples. The rule $R1$, whose coverage is shown in Figure 5.2(b), is extracted first because it covers the largest fraction of positive examples. All the training records covered by $R1$ are subsequently removed and the algorithm proceeds to look for the next best rule, which is $R2$.



(a) Original Data

(b) Step 1

(c) Step 2

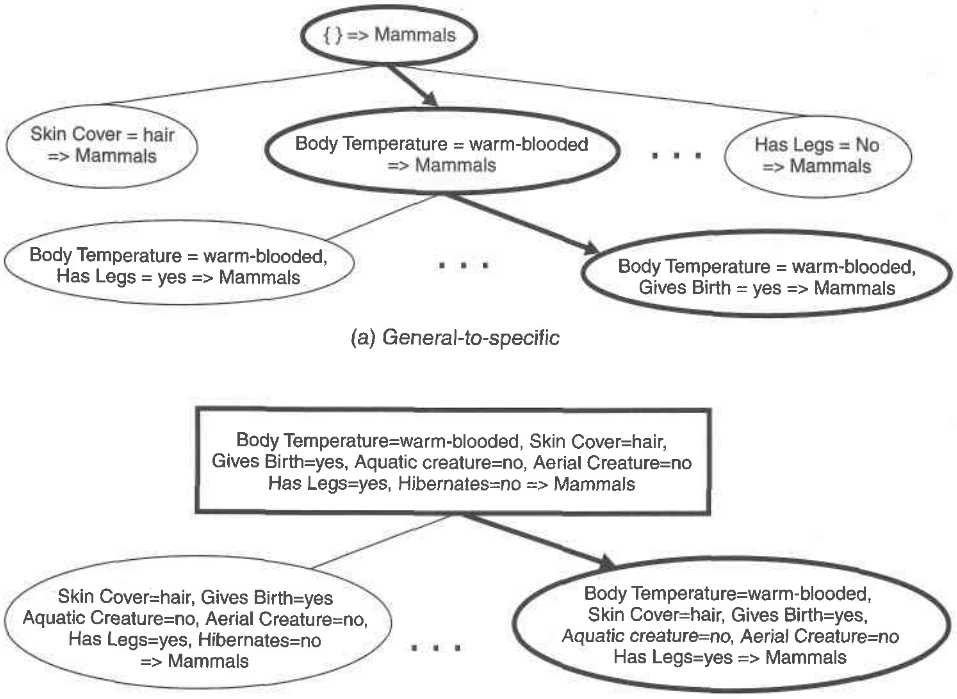
(d) Step 3

Figure 5.2. An example of the sequential covering algorithm.

Learn-One-Rule Function

The objective of the Learn-One-Rule function is to extract a classification rule that covers many of the positive examples and none (or very few) of the negative examples in the training set. However, finding an optimal rule is computationally expensive given the exponential size of the search space. The Learn-One-Rule function addresses the exponential search problem by growing the rules in a greedy fashion. It generates an initial rule r and keeps refining the rule until a certain stopping criterion is met. The rule is then pruned to improve its generalization error.

Rule-Growing Strategy There are two common strategies for growing a classification rule: general-to-specific or specific-to-general. Under the general-to-specific strategy, an initial rule $r : \{\} \longrightarrow y$ is created, where the left-hand side is an empty set and the right-hand side contains the target class. The rule has poor quality because it covers all the examples in the training set. New


{ } => Mammals
Skin Cover = hair => Mammals
Body Temperature = warm-blooded => Mammals
. . .
Has Legs = No => Mammals
Body Temperature = warm-blooded, Has Legs = yes => Mammals
. . .
Body Temperature = warm-blooded, Gives Birth = yes => Mammals
Body Temperature=warm-blooded, Skin Cover=hair, Gives Birth=yes, Aquatic creature=no, Aerial Creature=no Has Legs=yes, Hibernates=no => Mammals
Skin Cover=hair, Gives Birth=yes Aquatic Creature=no, Aerial Creature=no, Has Legs=yes, Hibernates=no => Mammals
. . .
Body Temperature=warm-blooded, Skin Cover=hair, Gives Birth=yes, Aquatic creature=no, Aerial Creature=no Has Legs=yes => Mammals

(a) General-to-specific

(b) Specific-to-general

Figure 5.3. General-to-specific and specific-to-general rule-growing strategies.

conjuncts are subsequently added to improve the rule's quality. Figure 5.3(a) shows the general-to-specific rule-growing strategy for the vertebrate classification problem. The conjunct `Body Temperature=warm-blooded` is initially chosen to form the rule antecedent. The algorithm then explores all the possible candidates and greedily chooses the next conjunct, `Gives Birth=yes`, to be added into the rule antecedent. This process continues until the stopping criterion is met (e.g., when the added conjunct does not improve the quality of the rule).

For the specific-to-general strategy, one of the positive examples is randomly chosen as the initial seed for the rule-growing process. During the refinement step, the rule is generalized by removing one of its conjuncts so that it can cover more positive examples. Figure 5.3(b) shows the specific-to-general approach for the vertebrate classification problem. Suppose a positive example for mammals is chosen as the initial seed. The initial rule contains the same conjuncts as the attribute values of the seed. To improve its coverage, the rule is generalized by removing the conjunct `Hibernate=no`. The refinement step is repeated until the stopping criterion is met, e.g., when the rule starts covering negative examples.

The previous approaches may produce suboptimal rules because the rules are grown in a greedy fashion. To avoid this problem, a beam search may be used, where k of the best candidate rules are maintained by the algorithm. Each candidate rule is then grown separately by adding (or removing) a conjunct from its antecedent. The quality of the candidates are evaluated and the k best candidates are chosen for the next iteration.

Rule Evaluation An evaluation metric is needed to determine which conjunct should be added (or removed) during the rule-growing process. Accuracy is an obvious choice because it explicitly measures the fraction of training examples classified correctly by the rule. However, a potential limitation of accuracy is that it does not take into account the rule's coverage. For example, consider a training set that contains 60 positive examples and 100 negative examples. Suppose we are given the following two candidate rules:

Rule r_1: covers 50 positive examples and 5 negative examples,
Rule r_2: covers 2 positive examples and no negative examples.

The accuracies for r_1 and r_2 are 90.9% and 100%, respectively. However, r_1 is the better rule despite its lower accuracy. The high accuracy for r_2 is potentially spurious because the coverage of the rule is too low.

The following approaches can be used to handle this problem.

1. A statistical test can be used to prune rules that have poor coverage. For example, we may compute the following likelihood ratio statistic:

   $$R = 2\sum_{i=1}^{k} f_i \log(f_i/e_i),$$

   where k is the number of classes, f_i is the observed frequency of class i examples that are covered by the rule, and e_i is the expected frequency of a rule that makes random predictions. Note that R has a chi-square distribution with $k-1$ degrees of freedom. A large R value suggests that the number of correct predictions made by the rule is significantly larger than that expected by random guessing. For example, since r_1 covers 55 examples, the expected frequency for the positive class is $e_+ = 55\times 60/160 = 20.625$, while the expected frequency for the negative class is $e_- = 55 \times 100/160 = 34.375$. Thus, the likelihood ratio for r_1 is

   $$R(r_1) = 2 \times [50 \times \log_2(50/20.625) + 5 \times \log_2(5/34.375)] = 99.9.$$

   Similarly, the expected frequencies for r_2 are $e_+ = 2 \times 60/160 = 0.75$ and $e_- = 2 \times 100/160 = 1.25$. The likelihood ratio statistic for r_2 is

   $$R(r_2) = 2 \times [2 \times \log_2(2/0.75) + 0 \times \log_2(0/1.25)] = 5.66.$$

   This statistic therefore suggests that r_1 is a better rule than r_2.

2. An evaluation metric that takes into account the rule coverage can be used. Consider the following evaluation metrics:

   $$\text{Laplace} = \frac{f_+ + 1}{n+k}, \tag{5.4}$$

   $$\text{m-estimate} = \frac{f_+ + kp_+}{n+k}, \tag{5.5}$$

   where n is the number of examples covered by the rule, f_+ is the number of positive examples covered by the rule, k is the total number of classes, and p_+ is the prior probability for the positive class. Note that the m-estimate is equivalent to the Laplace measure by choosing $p_+ = 1/k$. Depending on the rule coverage, these measures capture the trade-off between rule accuracy and the prior probability of the positive class. If the rule does not cover any training example, then the Laplace measure reduces to $1/k$, which is the prior probability of the positive class assuming a uniform class distribution. The m-estimate also reduces to the prior probability (p_+) when $n = 0$. However, if the rule coverage is large, then both measures asymptotically approach the rule accuracy, f_+/n. Going back to the previous example, the Laplace measure for r_1 is $51/57 = 89.47\%$, which is quite close to its accuracy. Conversely, the Laplace measure for r_2 (75%) is significantly lower than its accuracy because r_2 has a much lower coverage.

3. An evaluation metric that takes into account the support count of the rule can be used. One such metric is the **FOIL's information gain**. The support count of a rule corresponds to the number of positive examples covered by the rule. Suppose the rule $r: A \longrightarrow +$ covers p_0 positive examples and n_0 negative examples. After adding a new conjunct B, the extended rule $r': A \wedge B \longrightarrow +$ covers p_1 positive examples and n_1 negative examples. Given this information, the FOIL's information gain of the extended rule is defined as follows:

$$\text{FOIL's information gain} = p_1 \times \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right). \quad (5.6)$$

   Since the measure is proportional to p_1 and $p_1/(p_1 + n_1)$, it prefers rules that have high support count and accuracy. The FOIL's information gains for rules r_1 and r_2 given in the preceding example are 43.12 and 2, respectively. Therefore, r_1 is a better rule than r_2.

Rule Pruning The rules generated by the Learn-One-Rule function can be pruned to improve their generalization errors. To determine whether pruning is necessary, we may apply the methods described in Section 4.4 on page 172 to estimate the generalization error of a rule. For example, if the error on validation set decreases after pruning, we should keep the simplified rule. Another approach is to compare the pessimistic error of the rule before and after pruning (see Section 4.4.4 on page 179). The simplified rule is retained in place of the original rule if the pessimistic error improves after pruning.

Rationale for Sequential Covering

After a rule is extracted, the sequential covering algorithm must eliminate all the positive and negative examples covered by the rule. The rationale for doing this is given in the next example.

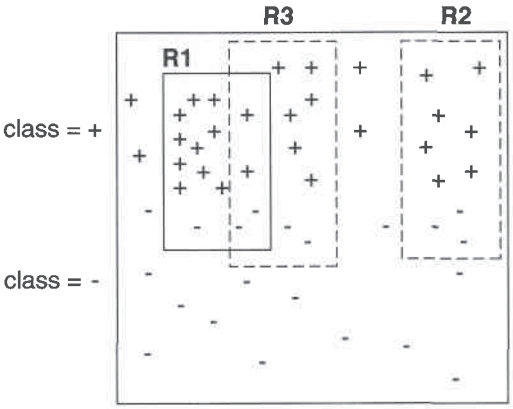


Figure 5.4. Elimination of training records by the sequential covering algorithm. $R1$, $R2$, and $R3$ represent regions covered by three different rules.

Figure 5.4 shows three possible rules, $R1$, $R2$, and $R3$, extracted from a data set that contains 29 positive examples and 21 negative examples. The accuracies of $R1$, $R2$, and $R3$ are 12/15 (80%), 7/10 (70%), and 8/12 (66.7%), respectively. $R1$ is generated first because it has the highest accuracy. After generating $R1$, it is clear that the positive examples covered by the rule must be removed so that the next rule generated by the algorithm is different than $R1$. Next, suppose the algorithm is given the choice of generating either $R2$ or $R3$. Even though $R2$ has higher accuracy than $R3$, $R1$ and $R3$ together cover 18 positive examples and 5 negative examples (resulting in an overall accuracy of 78.3%), whereas $R1$ and $R2$ together cover 19 positive examples and 6 negative examples (resulting in an overall accuracy of 76%). The incremental impact of $R2$ or $R3$ on accuracy is more evident when the positive and negative examples covered by $R1$ are removed before computing their accuracies. In particular, if positive examples covered by $R1$ are not removed, then we may overestimate the effective accuracy of $R3$, and if negative examples are not removed, then we may underestimate the accuracy of $R3$. In the latter case, we might end up preferring $R2$ over $R3$ even though half of the false positive errors committed by $R3$ have already been accounted for by the preceding rule, $R1$.

RIPPER Algorithm

To illustrate the direct method, we consider a widely used rule induction algorithm called RIPPER. This algorithm scales almost linearly with the number of training examples and is particularly suited for building models from data sets with imbalanced class distributions. RIPPER also works well with noisy data sets because it uses a validation set to prevent model overfitting.

For two-class problems, RIPPER chooses the majority class as its default class and learns the rules for detecting the minority class. For multiclass problems, the classes are ordered according to their frequencies. Let $(y_1, y_2, \ldots, y_c)$ be the ordered classes, where y_1 is the least frequent class and y_c is the most frequent class. During the first iteration, instances that belong to y_1 are labeled as positive examples, while those that belong to other classes are labeled as negative examples. The sequential covering method is used to generate rules that discriminate between the positive and negative examples. Next, RIPPER extracts rules that distinguish y_2 from other remaining classes. This process is repeated until we are left with y_c, which is designated as the default class.

Rule Growing RIPPER employs a general-to-specific strategy to grow a rule and the FOIL's information gain measure to choose the best conjunct to be added into the rule antecedent. It stops adding conjuncts when the rule starts covering negative examples. The new rule is then pruned based on its performance on the validation set. The following metric is computed to determine whether pruning is needed: $(p{-}n)/(p{+}n)$, where p (n) is the number of positive (negative) examples in the validation set covered by the rule. This metric is monotonically related to the rule's accuracy on the validation set. If the metric improves after pruning, then the conjunct is removed. Pruning is done starting from the last conjunct added to the rule. For example, given a rule $ABCD \longrightarrow y$, RIPPER checks whether D should be pruned first, followed by CD, BCD, etc. While the original rule covers only positive examples, the pruned rule may cover some of the negative examples in the training set.

Building the Rule Set After generating a rule, all the positive and negative examples covered by the rule are eliminated. The rule is then added into the rule set as long as it does not violate the stopping condition, which is based on the minimum description length principle. If the new rule increases the total description length of the rule set by at least d bits, then RIPPER stops adding rules into its rule set (by default, d is chosen to be 64 bits). Another stopping condition used by RIPPER is that the error rate of the rule on the validation set must not exceed 50%.

RIPPER also performs additional optimization steps to determine whether some of the existing rules in the rule set can be replaced by better alternative rules. Readers who are interested in the details of the optimization method may refer to the reference cited at the end of this chapter.

5.1.5 Indirect Methods for Rule Extraction

This section presents a method for generating a rule set from a decision tree. In principle, every path from the root node to the leaf node of a decision tree can be expressed as a classification rule. The test conditions encountered along the path form the conjuncts of the rule antecedent, while the class label at the leaf node is assigned to the rule consequent. Figure 5.5 shows an example of a rule set generated from a decision tree. Notice that the rule set is exhaustive and contains mutually exclusive rules. However, some of the rules can be simplified as shown in the next example.

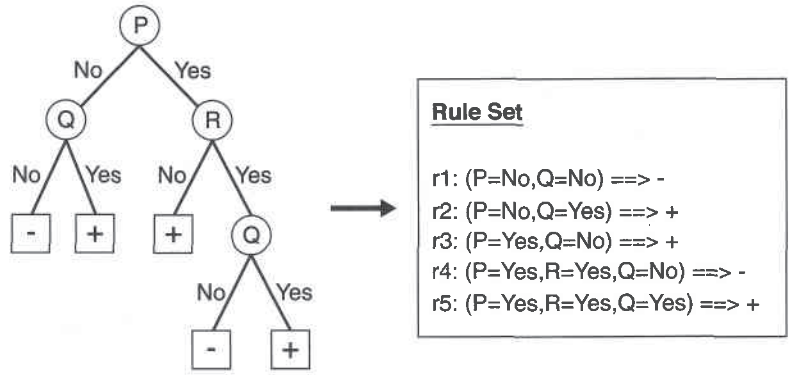


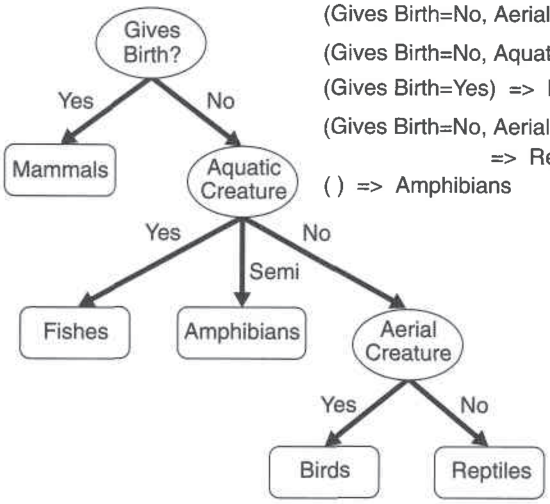
Figure 5.5. Converting a decision tree into classification rules.

Example 5.2. Consider the following three rules from Figure 5.5:

$r2 : (\text{P} = \text{No}) \wedge (\text{Q} = \text{Yes}) \longrightarrow +$
$r3 : (\text{P} = \text{Yes}) \wedge (\text{R} = \text{No}) \longrightarrow +$
$r5 : (\text{P} = \text{Yes}) \wedge (\text{R} = \text{Yes}) \wedge (\text{Q} = \text{Yes}) \longrightarrow +$

Observe that the rule set always predicts a positive class when the value of Q is Yes. Therefore, we may simplify the rules as follows:

$r2'$: $(\text{Q} = \text{Yes}) \longrightarrow +$
$r3$: $(\text{P} = \text{Yes}) \wedge (\text{R} = \text{No}) \longrightarrow +$.



Rule-Based Classifier:

(Gives Birth=No, Aerial Creature=Yes) => Birds

(Gives Birth=No, Aquatic Creature=Yes) => Fishes

(Gives Birth=Yes) => Mammals

(Gives Birth=No, Aerial Creature=No, Aquatic Creature=No)
=> Reptiles

( ) => Amphibians

Figure 5.6. Classification rules extracted from a decision tree for the vertebrate classification problem.

r_3 is retained to cover the remaining instances of the positive class. Although the rules obtained after simplification are no longer mutually exclusive, they are less complex and are easier to interpret. ■

In the following, we describe an approach used by the C4.5rules algorithm to generate a rule set from a decision tree. Figure 5.6 shows the decision tree and resulting classification rules obtained for the data set given in Table 5.2.

Rule Generation Classification rules are extracted for every path from the root to one of the leaf nodes in the decision tree. Given a classification rule $r : A \longrightarrow y$, we consider a simplified rule, $r' : A' \longrightarrow y$, where A' is obtained by removing one of the conjuncts in A. The simplified rule with the lowest pessimistic error rate is retained provided its error rate is less than that of the original rule. The rule-pruning step is repeated until the pessimistic error of the rule cannot be improved further. Because some of the rules may become identical after pruning, the duplicate rules must be discarded.

Rule Ordering After generating the rule set, C4.5rules uses the class-based ordering scheme to order the extracted rules. Rules that predict the same class are grouped together into the same subset. The total description length for each subset is computed, and the classes are arranged in increasing order of their total description length. The class that has the smallest description length is given the highest priority because it is expected to contain the best set of rules. The total description length for a class is given by $L_{\text{exception}} + g \times L_{\text{model}}$, where $L_{\text{exception}}$ is the number of bits needed to encode the misclassified examples, L_{model} is the number of bits needed to encode the model, and g is a tuning parameter whose default value is 0.5. The tuning parameter depends on the number of redundant attributes present in the model. The value of the tuning parameter is small if the model contains many redundant attributes.

5.1.6 Characteristics of Rule-Based Classifiers

A rule-based classifier has the following characteristics:

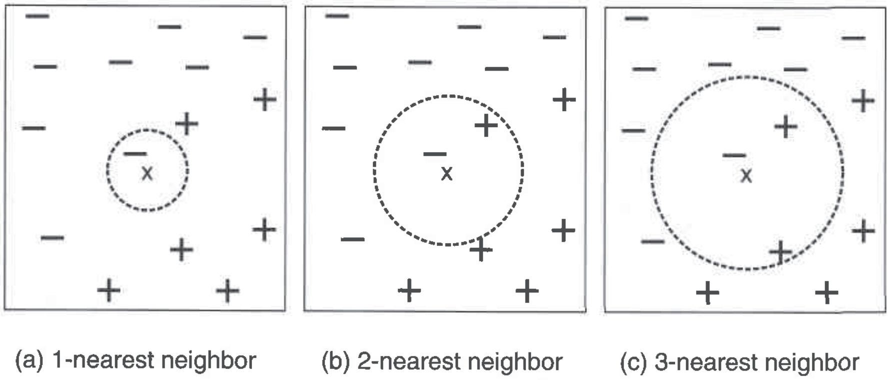
- The expressiveness of a rule set is almost equivalent to that of a decision tree because a decision tree can be represented by a set of mutually exclusive and exhaustive rules. Both rule-based and decision tree classifiers create rectilinear partitions of the attribute space and assign a class to each partition. Nevertheless, if the rule-based classifier allows multiple rules to be triggered for a given record, then a more complex decision boundary can be constructed.

- Rule-based classifiers are generally used to produce descriptive models that are easier to interpret, but gives comparable performance to the decision tree classifier.

- The class-based ordering approach adopted by many rule-based classifiers (such as RIPPER) is well suited for handling data sets with imbalanced class distributions.

5.2 Nearest-Neighbor classifiers

The classification framework shown in Figure 4.3 involves a two-step process: (1) an inductive step for constructing a classification model from data, and (2) a deductive step for applying the model to test examples. Decision tree and rule-based classifiers are examples of **eager learners** because they are designed to learn a model that maps the input attributes to the class label as soon as the training data becomes available. An opposite strategy would be to delay the process of modeling the training data until it is needed to classify the test examples. Techniques that employ this strategy are known as **lazy learners**. An example of a lazy learner is the **Rote classifier**, which memorizes the entire training data and performs classification only if the attributes of a test instance match one of the training examples exactly. An obvious drawback of



(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

Figure 5.7. The 1-, 2-, and 3-nearest neighbors of an instance.

this approach is that some test records may not be classified because they do not match any training example.

One way to make this approach more flexible is to find all the training examples that are relatively similar to the attributes of the test example. These examples, which are known as **nearest neighbors**, can be used to determine the class label of the test example. The justification for using nearest neighbors is best exemplified by the following saying: *"If it walks like a duck, quacks like a duck, and looks like a duck, then it's probably a duck."* A nearest-neighbor classifier represents each example as a data point in a d-dimensional space, where d is the number of attributes. Given a test example, we compute its proximity to the rest of the data points in the training set, using one of the proximity measures described in Section 2.4 on page 65. The k-nearest neighbors of a given example z refer to the k points that are closest to z.

Figure 5.7 illustrates the 1-, 2-, and 3-nearest neighbors of a data point located at the center of each circle. The data point is classified based on the class labels of its neighbors. In the case where the neighbors have more than one label, the data point is assigned to the majority class of its nearest neighbors. In Figure 5.7(a), the 1-nearest neighbor of the data point is a negative example. Therefore the data point is assigned to the negative class. If the number of nearest neighbors is three, as shown in Figure 5.7(c), then the neighborhood contains two positive examples and one negative example. Using the majority voting scheme, the data point is assigned to the positive class. In the case where there is a tie between the classes (see Figure 5.7(b)), we may randomly choose one of them to classify the data point.

The preceding discussion underscores the importance of choosing the right value for k. If k is too small, then the nearest-neighbor classifier may be

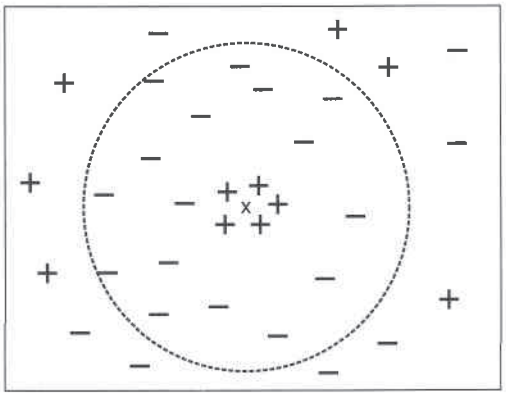


Figure 5.8. k-nearest neighbor classification with large k.

susceptible to overfitting because of noise in the training data. On the other hand, if k is too large, the nearest-neighbor classifier may misclassify the test instance because its list of nearest neighbors may include data points that are located far away from its neighborhood (see Figure 5.8).

5.2.1 Algorithm

A high-level summary of the nearest-neighbor classification method is given in Algorithm 5.2. The algorithm computes the distance (or similarity) between each test example $z = (\mathbf{x}', y')$ and all the training examples $(\mathbf{x}, y) \in D$ to determine its nearest-neighbor list, D_z. Such computation can be costly if the number of training examples is large. However, efficient indexing techniques are available to reduce the amount of computations needed to find the nearest neighbors of a test example.

Algorithm 5.2 The k-nearest neighbor classification algorithm.

```
Let $k$ be the number of nearest neighbors and $D$ be the set of training examples.
for each test example $z = (\mathbf{x}', y')$ do
    Compute $d(\mathbf{x}', \mathbf{x})$, the distance between $z$ and every example, $(\mathbf{x}, y) \in D$.
    Select $D_z \subseteq D$, the set of $k$ closest training examples to $z$.
    $y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} I(v = y_i)$
end for
```

Once the nearest-neighbor list is obtained, the test example is classified based on the majority class of its nearest neighbors:

$$\text{Majority Voting: } y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} I(v = y_i), \tag{5.7}$$

where v is a class label, y_i is the class label for one of the nearest neighbors, and $I(\cdot)$ is an indicator function that returns the value 1 if its argument is true and 0 otherwise.

In the majority voting approach, every neighbor has the same impact on the classification. This makes the algorithm sensitive to the choice of k, as shown in Figure 5.7. One way to reduce the impact of k is to weight the influence of each nearest neighbor $\mathbf{x}_i$ according to its distance: $w_i = 1/d(\mathbf{x}', \mathbf{x}_i)^2$. As a result, training examples that are located far away from z have a weaker impact on the classification compared to those that are located close to z. Using the distance-weighted voting scheme, the class label can be determined as follows:

$$\text{Distance-Weighted Voting: } y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} w_i \times I(v = y_i). \tag{5.8}$$

5.2.2 Characteristics of Nearest-Neighbor Classifiers

The characteristics of the nearest-neighbor classifier are summarized below:

- Nearest-neighbor classification is part of a more general technique known as instance-based learning, which uses specific training instances to make predictions without having to maintain an abstraction (or model) derived from data. Instance-based learning algorithms require a proximity measure to determine the similarity or distance between instances and a classification function that returns the predicted class of a test instance based on its proximity to other instances.

- Lazy learners such as nearest-neighbor classifiers do not require model building. However, classifying a test example can be quite expensive because we need to compute the proximity values individually between the test and training examples. In contrast, eager learners often spend the bulk of their computing resources for model building. Once a model has been built, classifying a test example is extremely fast.

- Nearest-neighbor classifiers make their predictions based on local information, whereas decision tree and rule-based classifiers attempt to find a global model that fits the entire input space. Because the classification decisions are made locally, nearest-neighbor classifiers (with small values of k) are quite susceptible to noise.

- Nearest-neighbor classifiers can produce arbitrarily shaped decision boundaries. Such boundaries provide a more flexible model representation compared to decision tree and rule-based classifiers that are often constrained to rectilinear decision boundaries. The decision boundaries of nearest-neighbor classifiers also have high variability because they depend on the composition of training examples. Increasing the number of nearest neighbors may reduce such variability.

- Nearest-neighbor classifiers can produce wrong predictions unless the appropriate proximity measure and data preprocessing steps are taken. For example, suppose we want to classify a group of people based on attributes such as height (measured in meters) and weight (measured in pounds). The height attribute has a low variability, ranging from 1.5 m to 1.85 m, whereas the weight attribute may vary from 90 lb. to 250 lb. If the scale of the attributes are not taken into consideration, the proximity measure may be dominated by differences in the weights of a person.

5.3 Bayesian Classifiers

In many applications the relationship between the attribute set and the class variable is non-deterministic. In other words, the class label of a test record cannot be predicted with certainty even though its attribute set is identical to some of the training examples. This situation may arise because of noisy data or the presence of certain confounding factors that affect classification but are not included in the analysis. For example, consider the task of predicting whether a person is at risk for heart disease based on the person's diet and workout frequency. Although most people who eat healthily and exercise regularly have less chance of developing heart disease, they may still do so because of other factors such as heredity, excessive smoking, and alcohol abuse. Determining whether a person's diet is healthy or the workout frequency is sufficient is also subject to interpretation, which in turn may introduce uncertainties into the learning problem.

This section presents an approach for modeling probabilistic relationships between the attribute set and the class variable. The section begins with an introduction to the **Bayes theorem**, a statistical principle for combining prior knowledge of the classes with new evidence gathered from data. The use of the Bayes theorem for solving classification problems will be explained, followed by a description of two implementations of Bayesian classifiers: naïve Bayes and the Bayesian belief network.

5.3.1 Bayes Theorem

> *Consider a football game between two rival teams: Team 0 and Team 1. Suppose Team 0 wins* 65% *of the time and Team 1 wins the remaining matches. Among the games won by Team 0, only* 30% *of them come from playing on Team 1's football field. On the other hand,* 75% *of the victories for Team 1 are obtained while playing at home. If Team 1 is to host the next match between the two teams, which team will most likely emerge as the winner?*

This question can be answered by using the well-known Bayes theorem. For completeness, we begin with some basic definitions from probability theory. Readers who are unfamiliar with concepts in probability may refer to Appendix C for a brief review of this topic.

Let X and Y be a pair of random variables. Their joint probability, $P(X = x, Y = y)$, refers to the probability that variable X will take on the value x and variable Y will take on the value y. A conditional probability is the probability that a random variable will take on a particular value given that the outcome for another random variable is known. For example, the conditional probability $P(Y = y|X = x)$ refers to the probability that the variable Y will take on the value y, given that the variable X is observed to have the value x. The joint and conditional probabilities for X and Y are related in the following way:

$$P(X, Y) = P(Y|X) \times P(X) = P(X|Y) \times P(Y). \tag{5.9}$$

Rearranging the last two expressions in Equation 5.9 leads to the following formula, known as the Bayes theorem:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}. \tag{5.10}$$

The Bayes theorem can be used to solve the prediction problem stated at the beginning of this section. For notational convenience, let X be the random variable that represents the team hosting the match and Y be the random variable that represents the winner of the match. Both X and Y can take on values from the set $\{0, 1\}$. We can summarize the information given in the problem as follows:

Probability Team 0 wins is $P(Y = 0) = 0.65$.
Probability Team 1 wins is $P(Y = 1) = 1 - P(Y = 0) = 0.35$.
Probability Team 1 hosted the match it won is $P(X = 1|Y = 1) = 0.75$.
Probability Team 1 hosted the match won by Team 0 is $P(X = 1|Y = 0) = 0.3$.

Our objective is to compute $P(Y = 1|X = 1)$, which is the conditional probability that Team 1 wins the next match it will be hosting, and compares it against $P(Y = 0|X = 1)$. Using the Bayes theorem, we obtain

$$\begin{aligned} P(Y=1|X=1) &= \frac{P(X=1|Y=1) \times P(Y=1)}{P(X=1)} \\ &= \frac{P(X=1|Y=1) \times P(Y=1)}{P(X=1,Y=1) + P(X=1,Y=0)} \\ &= \frac{P(X=1|Y=1) \times P(Y=1)}{P(X=1|Y=1)P(Y=1) + P(X=1|Y=0)P(Y=0)} \\ &= \frac{0.75 \times 0.35}{0.75 \times 0.35 + 0.3 \times 0.65} \\ &= 0.5738, \end{aligned}$$

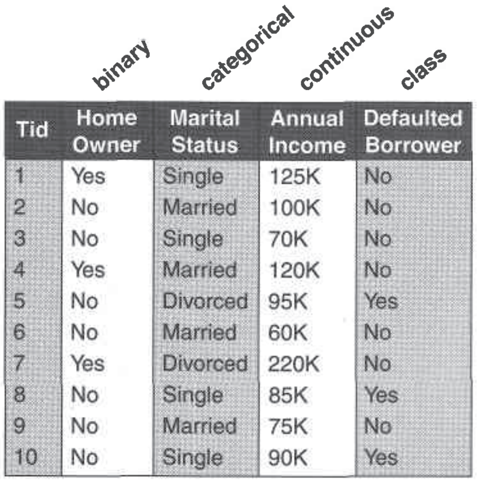
where the law of total probability (see Equation C.5 on page 722) was applied in the second line. Furthermore, $P(Y = 0|X = 1) = 1 - P(Y = 1|X = 1) = 0.4262$. Since $P(Y = 1|X = 1) > P(Y = 0|X = 1)$, Team 1 has a better chance than Team 0 of winning the next match.

5.3.2 Using the Bayes Theorem for Classification

Before describing how the Bayes theorem can be used for classification, let us formalize the classification problem from a statistical perspective. Let $\mathbf{X}$ denote the attribute set and Y denote the class variable. If the class variable has a non-deterministic relationship with the attributes, then we can treat $\mathbf{X}$ and Y as random variables and capture their relationship probabilistically using $P(Y|\mathbf{X})$. This conditional probability is also known as the **posterior probability** for Y, as opposed to its **prior probability**, $P(Y)$.

During the training phase, we need to learn the posterior probabilities $P(Y|\mathbf{X})$ for every combination of $\mathbf{X}$ and Y based on information gathered from the training data. By knowing these probabilities, a test record $\mathbf{X}'$ can be classified by finding the class Y' that maximizes the posterior probability,

$P(Y'|\mathbf{X}')$. To illustrate this approach, consider the task of predicting whether a loan borrower will default on their payments. Figure 5.9 shows a training set with the following attributes: `Home Owner`, `Marital Status`, and `Annual Income`. Loan borrowers who defaulted on their payments are classified as `Yes`, while those who repaid their loans are classified as `No`.


binary
categorical
continuous
class

Tid	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Figure 5.9. Training set for predicting the loan default problem.

Suppose we are given a test record with the following attribute set: $\mathbf{X} =$ (`Home Owner` = `No`, `Marital Status` = `Married`, `Annual Income` = \$120K). To classify the record, we need to compute the posterior probabilities $P(\texttt{Yes}|\mathbf{X})$ and $P(\texttt{No}|\mathbf{X})$ based on information available in the training data. If $P(\texttt{Yes}|\mathbf{X}) > P(\texttt{No}|\mathbf{X})$, then the record is classified as `Yes`; otherwise, it is classified as `No`.

Estimating the posterior probabilities accurately for every possible combination of class label and attribute value is a difficult problem because it requires a very large training set, even for a moderate number of attributes. The Bayes theorem is useful because it allows us to express the posterior probability in terms of the prior probability $P(Y)$, the **class-conditional** probability $P(\mathbf{X}|Y)$, and the evidence, $P(\mathbf{X})$:

$$P(Y|\mathbf{X}) = \frac{P(\mathbf{X}|Y) \times P(Y)}{P(\mathbf{X})}. \tag{5.11}$$

When comparing the posterior probabilities for different values of Y, the denominator term, $P(\mathbf{X})$, is always constant, and thus, can be ignored. The prior probability $P(Y)$ can be easily estimated from the training set by computing the fraction of training records that belong to each class. To estimate the class-conditional probabilities $P(\mathbf{X}|Y)$, we present two implementations of Bayesian classification methods: the naïve Bayes classifier and the Bayesian belief network. These implementations are described in Sections 5.3.3 and 5.3.5, respectively.

5.3.3 Naïve Bayes Classifier

A naïve Bayes classifier estimates the class-conditional probability by assuming that the attributes are conditionally independent, given the class label y. The conditional independence assumption can be formally stated as follows:

$$P(\mathbf{X}|Y = y) = \prod_{i=1}^{d} P(X_i|Y = y), \tag{5.12}$$

where each attribute set $\mathbf{X} = \{X_1, X_2, \ldots, X_d\}$ consists of d attributes.

Conditional Independence

Before delving into the details of how a naïve Bayes classifier works, let us examine the notion of conditional independence. Let $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ denote three sets of random variables. The variables in $\mathbf{X}$ are said to be conditionally independent of $\mathbf{Y}$, given $\mathbf{Z}$, if the following condition holds:

$$P(\mathbf{X}|\mathbf{Y}, \mathbf{Z}) = P(\mathbf{X}|\mathbf{Z}). \tag{5.13}$$

An example of conditional independence is the relationship between a person's arm length and his or her reading skills. One might observe that people with longer arms tend to have higher levels of reading skills. This relationship can be explained by the presence of a confounding factor, which is age. A young child tends to have short arms and lacks the reading skills of an adult. If the age of a person is fixed, then the observed relationship between arm length and reading skills disappears. Thus, we can conclude that arm length and reading skills are conditionally independent when the age variable is fixed.

The conditional independence between $\mathbf{X}$ and $\mathbf{Y}$ can also be written into a form that looks similar to Equation 5.12:

$$\begin{aligned}
P(\mathbf{X},\mathbf{Y}|\mathbf{Z}) &= \frac{P(\mathbf{X},\mathbf{Y},\mathbf{Z})}{P(\mathbf{Z})} \\
&= \frac{P(\mathbf{X},\mathbf{Y},\mathbf{Z})}{P(\mathbf{Y},\mathbf{Z})} \times \frac{P(\mathbf{Y},\mathbf{Z})}{P(\mathbf{Z})} \\
&= P(\mathbf{X}|\mathbf{Y},\mathbf{Z}) \times P(\mathbf{Y}|\mathbf{Z}) \\
&= P(\mathbf{X}|\mathbf{Z}) \times P(\mathbf{Y}|\mathbf{Z}), \qquad (5.14)
\end{aligned}$$

where Equation 5.13 was used to obtain the last line of Equation 5.14.

How a Naïve Bayes Classifier Works

With the conditional independence assumption, instead of computing the class-conditional probability for every combination of $\mathbf{X}$, we only have to estimate the conditional probability of each X_i, given Y. The latter approach is more practical because it does not require a very large training set to obtain a good estimate of the probability.

To classify a test record, the naïve Bayes classifier computes the posterior probability for each class Y:

$$P(Y|\mathbf{X}) = \frac{P(Y)\prod_{i=1}^{d} P(X_i|Y)}{P(\mathbf{X})}. \qquad (5.15)$$

Since $P(\mathbf{X})$ is fixed for every Y, it is sufficient to choose the class that maximizes the numerator term, $P(Y)\prod_{i=1}^{d} P(X_i|Y)$. In the next two subsections, we describe several approaches for estimating the conditional probabilities $P(X_i|Y)$ for categorical and continuous attributes.

Estimating Conditional Probabilities for Categorical Attributes

For a categorical attribute X_i, the conditional probability $P(X_i = x_i|Y = y)$ is estimated according to the fraction of training instances in class y that take on a particular attribute value x_i. For example, in the training set given in Figure 5.9, three out of the seven people who repaid their loans also own a home. As a result, the conditional probability for $P(\texttt{Home Owner}=\texttt{Yes}|\texttt{No})$ is equal to 3/7. Similarly, the conditional probability for defaulted borrowers who are single is given by $P(\texttt{Marital Status} = \texttt{Single}|\texttt{Yes}) = 2/3$.

Estimating Conditional Probabilities for Continuous Attributes

There are two ways to estimate the class-conditional probabilities for continuous attributes in naïve Bayes classifiers:

1. We can discretize each continuous attribute and then replace the continuous attribute value with its corresponding discrete interval. This approach transforms the continuous attributes into ordinal attributes. The conditional probability $P(X_i|Y = y)$ is estimated by computing the fraction of training records belonging to class y that falls within the corresponding interval for X_i. The estimation error depends on the discretization strategy (as described in Section 2.3.6 on page 57), as well as the number of discrete intervals. If the number of intervals is too large, there are too few training records in each interval to provide a reliable estimate for $P(X_i|Y)$. On the other hand, if the number of intervals is too small, then some intervals may aggregate records from different classes and we may miss the correct decision boundary.

2. We can assume a certain form of probability distribution for the continuous variable and estimate the parameters of the distribution using the training data. A Gaussian distribution is usually chosen to represent the class-conditional probability for continuous attributes. The distribution is characterized by two parameters, its mean, μ, and variance, σ^2. For each class y_j, the class-conditional probability for attribute X_i is

$$P(X_i = x_i|Y = y_j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} \exp^{-\frac{(x_i-\mu_{ij})^2}{2\sigma_{ij}^2}}. \tag{5.16}$$

   The parameter μ_{ij} can be estimated based on the sample mean of X_i ($\overline{x}$) for all training records that belong to the class y_j. Similarly, σ_{ij}^2 can be estimated from the sample variance (s^2) of such training records. For example, consider the annual income attribute shown in Figure 5.9. The sample mean and variance for this attribute with respect to the class `No` are

$$\overline{x} = \frac{125 + 100 + 70 + \ldots + 75}{7} = 110$$
$$s^2 = \frac{(125 - 110)^2 + (100 - 110)^2 + \ldots + (75 - 110)^2}{7(6)} = 2975$$
$$s = \sqrt{2975} = 54.54.$$

Given a test record with taxable income equal to $120K, we can compute its class-conditional probability as follows:

$$P(\texttt{Income=120|No}) = \frac{1}{\sqrt{2\pi}(54.54)} \exp^{-\frac{(120-110)^2}{2\times 2975}} = 0.0072.$$

Note that the preceding interpretation of class-conditional probability is somewhat misleading. The right-hand side of Equation 5.16 corresponds to a **probability density function**, $f(X_i; \mu_{ij}, \sigma_{ij})$. Since the function is continuous, the probability that the random variable X_i takes a particular value is zero. Instead, we should compute the conditional probability that X_i lies within some interval, x_i and $x_i + \epsilon$, where ϵ is a small constant:

$$\begin{aligned} P(x_i \le X_i \le x_i + \epsilon | Y = y_j) &= \int_{x_i}^{x_i+\epsilon} f(X_i; \mu_{ij}, \sigma_{ij}) dX_i \\ &\approx f(x_i; \mu_{ij}, \sigma_{ij}) \times \epsilon. \qquad (5.17) \end{aligned}$$

Since ϵ appears as a constant multiplicative factor for each class, it cancels out when we normalize the posterior probability for $P(Y|\mathbf{X})$. Therefore, we can still apply Equation 5.16 to approximate the class-conditional probability $P(X_i|Y)$.

Example of the Naïve Bayes Classifier

Consider the data set shown in Figure 5.10(a). We can compute the class-conditional probability for each categorical attribute, along with the sample mean and variance for the continuous attribute using the methodology described in the previous subsections. These probabilities are summarized in Figure 5.10(b).

To predict the class label of a test record $\mathbf{X}$ = (Home Owner=No, Marital Status = Married, Income = $120K), we need to compute the posterior probabilities $P(\texttt{No}|\mathbf{X})$ and $P(\texttt{Yes}|\mathbf{X})$. Recall from our earlier discussion that these posterior probabilities can be estimated by computing the product between the prior probability $P(Y)$ and the class-conditional probabilities $\prod_i P(X_i|Y)$, which corresponds to the numerator of the right-hand side term in Equation 5.15.

The prior probabilities of each class can be estimated by calculating the fraction of training records that belong to each class. Since there are three records that belong to the class Yes and seven records that belong to the class

Tid	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

(a)

P(Home Owner=Yes|No) = 3/7
P(Home Owner=No|No) = 4/7
P(Home Owner=Yes|Yes) = 0
P(Home Owner=No|Yes) = 1
P(Marital Status=Single|No) = 2/7
P(Marital Status=Divorced|No) = 1/7
P(Marital Status=Married|No) = 4/7
P(Marital Status=Single|Yes) = 2/3
P(Marital Status=Divorced|Yes) = 1/3
P(Marital Status=Married|Yes) = 0

For Annual Income:
If class=No: sample mean=110
sample variance=2975
If class=Yes: sample mean=90
sample variance=25

(b)

Figure 5.10. The naïve Bayes classifier for the loan classification problem.

No, $P(\texttt{Yes}) = 0.3$ and $P(\texttt{No}) = 0.7$. Using the information provided in Figure 5.10(b), the class-conditional probabilities can be computed as follows:

$$\begin{aligned} P(\mathbf{X}|\texttt{No}) &= P(\texttt{Home Owner} = \texttt{No}|\texttt{No}) \times P(\texttt{Status} = \texttt{Married}|\texttt{No}) \\ &\quad \times P(\texttt{Annual Income} = \$120\text{K}|\texttt{No}) \\ &= 4/7 \times 4/7 \times 0.0072 = 0.0024. \end{aligned}$$

$$\begin{aligned} P(\mathbf{X}|\texttt{Yes}) &= P(\texttt{Home Owner} = \texttt{No}|\texttt{Yes}) \times P(\texttt{Status} = \texttt{Married}|\texttt{Yes}) \\ &\quad \times P(\texttt{Annual Income} = \$120\text{K}|\texttt{Yes}) \\ &= 1 \times 0 \times 1.2 \times 10^{-9} = 0. \end{aligned}$$

Putting them together, the posterior probability for class No is $P(\texttt{No}|\mathbf{X}) = \alpha \times 7/10 \times 0.0024 = 0.0016\alpha$, where $\alpha = 1/P(\mathbf{X})$ is a constant term. Using a similar approach, we can show that the posterior probability for class Yes is zero because its class-conditional probability is zero. Since $P(\texttt{No}|\mathbf{X}) > P(\texttt{Yes}|\mathbf{X})$, the record is classified as No.

M-estimate of Conditional Probability

The preceding example illustrates a potential problem with estimating posterior probabilities from training data. If the class-conditional probability for one of the attributes is zero, then the overall posterior probability for the class vanishes. This approach of estimating class-conditional probabilities using simple fractions may seem too brittle, especially when there are few training examples available and the number of attributes is large.

In a more extreme case, if the training examples do not cover many of the attribute values, we may not be able to classify some of the test records. For example, if $P(\texttt{Marital Status} = \texttt{Divorced}|\texttt{No})$ is zero instead of 1/7, then a record with attribute set $\mathbf{X} = (\texttt{Home Owner} = \texttt{Yes}, \texttt{Marital Status} = \texttt{Divorced}, \texttt{Income} = \$120\text{K})$ has the following class-conditional probabilities:

$$P(\mathbf{X}|\texttt{No}) = 3/7 \times 0 \times 0.0072 = 0.$$
$$P(\mathbf{X}|\texttt{Yes}) = 0 \times 1/3 \times 1.2 \times 10^{-9} = 0.$$

The naïve Bayes classifier will not be able to classify the record. This problem can be addressed by using the m-estimate approach for estimating the conditional probabilities:

$$P(x_i|y_j) = \frac{n_c + mp}{n + m}, \tag{5.18}$$

where n is the total number of instances from class y_j, n_c is the number of training examples from class y_j that take on the value x_i, m is a parameter known as the equivalent sample size, and p is a user-specified parameter. If there is no training set available (i.e., $n = 0$), then $P(x_i|y_j) = p$. Therefore p can be regarded as the prior probability of observing the attribute value x_i among records with class y_j. The equivalent sample size determines the tradeoff between the prior probability p and the observed probability n_c/n.

In the example given in the previous section, the conditional probability $P(\texttt{Status} = \texttt{Married}|\texttt{Yes}) = 0$ because none of the training records for the class has the particular attribute value. Using the m-estimate approach with $m = 3$ and $p = 1/3$, the conditional probability is no longer zero:

$$P(\texttt{Marital Status} = \texttt{Married}|\texttt{Yes}) = (0 + 3 \times 1/3)/(3 + 3) = 1/6.$$

If we assume $p = 1/3$ for all attributes of class Yes and $p = 2/3$ for all attributes of class No, then

$$\begin{aligned}
P(\mathbf{X}|\texttt{No}) &= P(\texttt{Home Owner} = \texttt{No}|\texttt{No}) \times P(\texttt{Status} = \texttt{Married}|\texttt{No}) \\
&\quad \times P(\texttt{Annual Income} = \$120\text{K}|\texttt{No}) \\
&= 6/10 \times 6/10 \times 0.0072 = 0.0026.
\end{aligned}$$

$$\begin{aligned}
P(\mathbf{X}|\texttt{Yes}) &= P(\texttt{Home Owner} = \texttt{No}|\texttt{Yes}) \times P(\texttt{Status} = \texttt{Married}|\texttt{Yes}) \\
&\quad \times P(\texttt{Annual Income} = \$120\text{K}|\texttt{Yes}) \\
&= 4/6 \times 1/6 \times 1.2 \times 10^{-9} = 1.3 \times 10^{-10}.
\end{aligned}$$

The posterior probability for class No is $P(\texttt{No}|\mathbf{X}) = \alpha \times 7/10 \times 0.0026 = 0.0018\alpha$, while the posterior probability for class Yes is $P(\texttt{Yes}|\mathbf{X}) = \alpha \times 3/10 \times 1.3 \times 10^{-10} = 4.0 \times 10^{-11}\alpha$. Although the classification decision has not changed, the m-estimate approach generally provides a more robust way for estimating probabilities when the number of training examples is small.

Characteristics of Naïve Bayes Classifiers

Naïve Bayes classifiers generally have the following characteristics:

- They are robust to isolated noise points because such points are averaged out when estimating conditional probabilities from data. Naïve Bayes classifiers can also handle missing values by ignoring the example during model building and classification.

- They are robust to irrelevant attributes. If X_i is an irrelevant attribute, then $P(X_i|Y)$ becomes almost uniformly distributed. The class-conditional probability for X_i has no impact on the overall computation of the posterior probability.

- Correlated attributes can degrade the performance of naïve Bayes classifiers because the conditional independence assumption no longer holds for such attributes. For example, consider the following probabilities:

$$\begin{aligned}
P(A = 0|Y = 0) = 0.4, \quad P(A = 1|Y = 0) = 0.6, \\
P(A = 0|Y = 1) = 0.6, \quad P(A = 1|Y = 1) = 0.4,
\end{aligned}$$

  where A is a binary attribute and Y is a binary class variable. Suppose there is another binary attribute B that is perfectly correlated with A

when $Y = 0$, but is independent of A when $Y = 1$. For simplicity, assume that the class-conditional probabilities for B are the same as for A. Given a record with attributes $A = 0, B = 0$, we can compute its posterior probabilities as follows:

$$\begin{aligned} P(Y = 0|A = 0, B = 0) &= \frac{P(A = 0|Y = 0)P(B = 0|Y = 0)P(Y = 0)}{P(A = 0, B = 0)} \\ &= \frac{0.16 \times P(Y = 0)}{P(A = 0, B = 0)}. \end{aligned}$$

$$\begin{aligned} P(Y = 1|A = 0, B = 0) &= \frac{P(A = 0|Y = 1)P(B = 0|Y = 1)P(Y = 1)}{P(A = 0, B = 0)} \\ &= \frac{0.36 \times P(Y = 1)}{P(A = 0, B = 0)}. \end{aligned}$$

If $P(Y = 0) = P(Y = 1)$, then the naïve Bayes classifier would assign the record to class 1. However, the truth is,

$$P(A = 0, B = 0|Y = 0) = P(A = 0|Y = 0) = 0.4,$$

because A and B are perfectly correlated when $Y = 0$. As a result, the posterior probability for $Y = 0$ is

$$\begin{aligned} P(Y = 0|A = 0, B = 0) &= \frac{P(A = 0, B = 0|Y = 0)P(Y = 0)}{P(A = 0, B = 0)} \\ &= \frac{0.4 \times P(Y = 0)}{P(A = 0, B = 0)}, \end{aligned}$$

which is larger than that for $Y = 1$. The record should have been classified as class 0.

5.3.4 Bayes Error Rate

Suppose we know the true probability distribution that governs $P(\mathbf{X}|Y)$. The Bayesian classification method allows us to determine the ideal decision boundary for the classification task, as illustrated in the following example.

Example 5.3. Consider the task of identifying alligators and crocodiles based on their respective lengths. The average length of an adult crocodile is about 15 feet, while the average length of an adult alligator is about 12 feet. Assuming

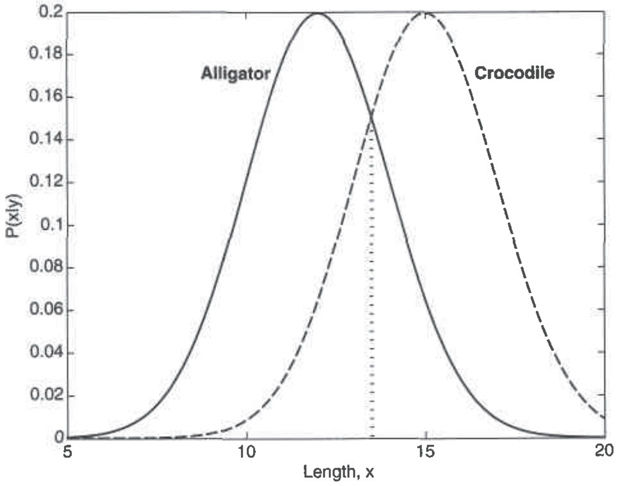


Figure 5.11. Comparing the likelihood functions of a crocodile and an alligator.

that their length x follows a Gaussian distribution with a standard deviation equal to 2 feet, we can express their class-conditional probabilities as follows:

$$P(X|\texttt{Crocodile}) = \frac{1}{\sqrt{2\pi}\cdot 2}\exp\left[-\frac{1}{2}\left(\frac{X-15}{2}\right)^2\right] \qquad (5.19)$$

$$P(X|\texttt{Alligator}) = \frac{1}{\sqrt{2\pi}\cdot 2}\exp\left[-\frac{1}{2}\left(\frac{X-12}{2}\right)^2\right] \qquad (5.20)$$

Figure 5.11 shows a comparison between the class-conditional probabilities for a crocodile and an alligator. Assuming that their prior probabilities are the same, the ideal decision boundary is located at some length $\hat{x}$ such that

$$P(X=\hat{x}|\texttt{Crocodile}) = P(X=\hat{x}|\texttt{Alligator}).$$

Using Equations 5.19 and 5.20, we obtain

$$\left(\frac{\hat{x}-15}{2}\right)^2 = \left(\frac{\hat{x}-12}{2}\right)^2,$$

which can be solved to yield $\hat{x} = 13.5$. The decision boundary for this example is located halfway between the two means. ■

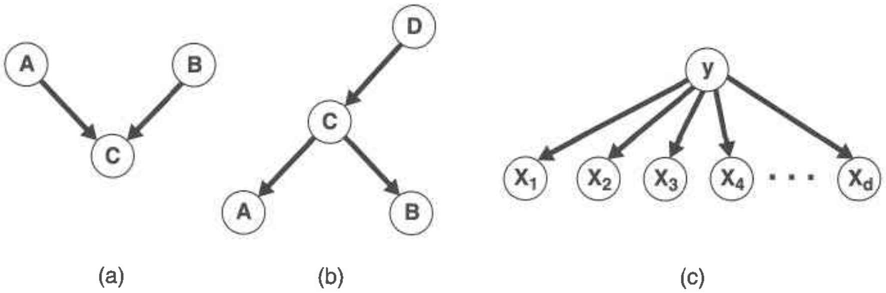


Figure 5.12. Representing probabilistic relationships using directed acyclic graphs.

When the prior probabilities are different, the decision boundary shifts toward the class with lower prior probability (see Exercise 10 on page 319). Furthermore, the minimum error rate attainable by any classifier on the given data can also be computed. The ideal decision boundary in the preceding example classifies all creatures whose lengths are less than $\hat{x}$ as alligators and those whose lengths are greater than $\hat{x}$ as crocodiles. The error rate of the classifier is given by the sum of the area under the posterior probability curve for crocodiles (from length 0 to $\hat{x}$) and the area under the posterior probability curve for alligators (from $\hat{x}$ to ∞):

$$\text{Error} = \int_0^{\hat{x}} P(\texttt{Crocodile}|X)dX + \int_{\hat{x}}^{\infty} P(\texttt{Alligator}|X)dX.$$

The total error rate is known as the **Bayes error rate**.

5.3.5 Bayesian Belief Networks

The conditional independence assumption made by naïve Bayes classifiers may seem too rigid, especially for classification problems in which the attributes are somewhat correlated. This section presents a more flexible approach for modeling the class-conditional probabilities $P(\mathbf{X}|Y)$. Instead of requiring all the attributes to be conditionally independent given the class, this approach allows us to specify which pair of attributes are conditionally independent. We begin with a discussion on how to represent and build such a probabilistic model, followed by an example of how to make inferences from the model.

Model Representation

A Bayesian belief network (BBN), or simply, Bayesian network, provides a graphical representation of the probabilistic relationships among a set of random variables. There are two key elements of a Bayesian network:

1. A directed acyclic graph (dag) encoding the dependence relationships among a set of variables.

2. A probability table associating each node to its immediate parent nodes.

Consider three random variables, A, B, and C, in which A and B are independent variables and each has a direct influence on a third variable, C. The relationships among the variables can be summarized into the directed acyclic graph shown in Figure 5.12(a). Each node in the graph represents a variable, and each arc asserts the dependence relationship between the pair of variables. If there is a directed arc from X to Y, then X is the **parent** of Y and Y is the **child** of X. Furthermore, if there is a directed path in the network from X to Z, then X is an **ancestor** of Z, while Z is a **descendant** of X. For example, in the diagram shown in Figure 5.12(b), A is a descendant of D and D is an ancestor of B. Both B and D are also non-descendants of A. An important property of the Bayesian network can be stated as follows:

Property 1 (Conditional Independence). *A node in a Bayesian network is conditionally independent of its non-descendants, if its parents are known.*

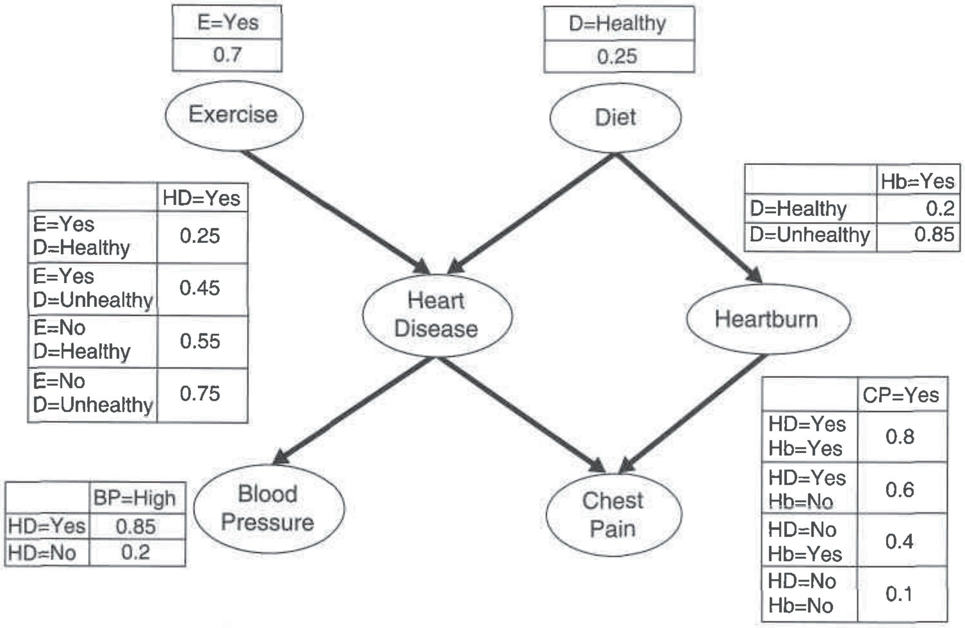
In the diagram shown in Figure 5.12(b), A is conditionally independent of both B and D given C because the nodes for B and D are non-descendants of node A. The conditional independence assumption made by a naïve Bayes classifier can also be represented using a Bayesian network, as shown in Figure 5.12(c), where y is the target class and $\{X_1, X_2, \ldots, X_d\}$ is the attribute set.

Besides the conditional independence conditions imposed by the network topology, each node is also associated with a probability table.

1. If a node X does not have any parents, then the table contains only the prior probability $P(X)$.

2. If a node X has only one parent, Y, then the table contains the conditional probability $P(X|Y)$.

3. If a node X has multiple parents, $\{Y_1, Y_2, \ldots, Y_k\}$, then the table contains the conditional probability $P(X|Y_1, Y_2, \ldots, Y_k)$.



E=Yes
0.7

D=Healthy
0.25

	HD=Yes

E=Yes
D=Healthy
	0.25

E=Yes
D=Unhealthy
	0.45

E=No
D=Healthy
	0.55

E=No
D=Unhealthy
	0.75

	Hb=Yes
D=Healthy	0.2
D=Unhealthy	0.85

	BP=High
HD=Yes	0.85
HD=No	0.2

	CP=Yes

HD=Yes
Hb=Yes
	0.8

HD=Yes
Hb=No
	0.6

HD=No
Hb=Yes
	0.4

HD=No
Hb=No
	0.1

Figure 5.13. A Bayesian belief network for detecting heart disease and heartburn in patients.

Figure 5.13 shows an example of a Bayesian network for modeling patients with heart disease or heartburn problems. Each variable in the diagram is assumed to be binary-valued. The parent nodes for heart disease (HD) correspond to risk factors that may affect the disease, such as exercise (E) and diet (D). The child nodes for heart disease correspond to symptoms of the disease, such as chest pain (CP) and high blood pressure (BP). For example, the diagram shows that heartburn (Hb) may result from an unhealthy diet and may lead to chest pain.

The nodes associated with the risk factors contain only the prior probabilities, whereas the nodes for heart disease, heartburn, and their corresponding symptoms contain the conditional probabilities. To save space, some of the probabilities have been omitted from the diagram. The omitted probabilities can be recovered by noting that $P(X = \overline{x}) = 1 - P(X = x)$ and $P(X = \overline{x}|Y) = 1 - P(X = x|Y)$, where $\overline{x}$ denotes the opposite outcome of x. For example, the conditional probability

$$\begin{aligned}
& P(\texttt{Heart Disease} = \texttt{No}|\texttt{Exercise} = \texttt{No}, \texttt{Diet} = \texttt{Healthy}) \\
= \; & 1 - P(\texttt{Heart Disease} = \texttt{Yes}|\texttt{Exercise} = \texttt{No}, \texttt{Diet} = \texttt{Healthy}) \\
= \; & 1 - 0.55 = 0.45.
\end{aligned}$$

Model Building

Model building in Bayesian networks involves two steps: (1) creating the structure of the network, and (2) estimating the probability values in the tables associated with each node. The network topology can be obtained by encoding the subjective knowledge of domain experts. Algorithm 5.3 presents a systematic procedure for inducing the topology of a Bayesian network.

Algorithm 5.3 Algorithm for generating the topology of a Bayesian network.

```
1: Let $T = (X_1, X_2, \ldots, X_d)$ denote a total order of the variables.
2: for $j = 1$ to $d$ do
3:    Let $X_{T(j)}$ denote the $j^{th}$ highest order variable in $T$.
4:    Let $\pi(X_{T(j)}) = \{X_{T(1)}, X_{T(2)}, \ldots, X_{T(j-1)}\}$ denote the set of variables preceding $X_{T(j)}$.
5:    Remove the variables from $\pi(X_{T(j)})$ that do not affect $X_j$ (using prior knowledge).
6:    Create an arc between $X_{T(j)}$ and the remaining variables in $\pi(X_{T(j)})$.
7: end for
```

Example 5.4. Consider the variables shown in Figure 5.13. After performing Step 1, let us assume that the variables are ordered in the following way: (E, D, HD, Hb, CP, BP). From Steps 2 to 7, starting with variable D, we obtain the following conditional probabilities:

- $P(D|E)$ is simplified to $P(D)$.
- $P(HD|E, D)$ cannot be simplified.
- $P(Hb|HD, E, D)$ is simplified to $P(Hb|D)$.
- $P(CP|Hb, HD, E, D)$ is simplified to $P(CP|Hb, HD)$.
- $P(BP|CP, Hb, HD, E, D)$ is simplified to $P(BP|HD)$.

Based on these conditional probabilities, we can create arcs between the nodes (E, HD), (D, HD), (D, Hb), (HD, CP), (Hb, CP), and (HD, BP). These arcs result in the network structure shown in Figure 5.13. ■

Algorithm 5.3 guarantees a topology that does not contain any cycles. The proof for this is quite straightforward. If a cycle exists, then there must be at least one arc connecting the lower-ordered nodes to the higher-ordered nodes, and at least another arc connecting the higher-ordered nodes to the lower-ordered nodes. Since Algorithm 5.3 prevents any arc from connecting the lower-ordered nodes to the higher-ordered nodes, there cannot be any cycles in the topology.

Nevertheless, the network topology may change if we apply a different ordering scheme to the variables. Some topology may be inferior because it produces many arcs connecting between different pairs of nodes. In principle, we may have to examine all $d!$ possible orderings to determine the most appropriate topology, a task that can be computationally expensive. An alternative approach is to divide the variables into causal and effect variables, and then draw the arcs from each causal variable to its corresponding effect variables. This approach eases the task of building the Bayesian network structure.

Once the right topology has been found, the probability table associated with each node is determined. Estimating such probabilities is fairly straightforward and is similar to the approach used by naïve Bayes classifiers.

Example of Inferencing Using BBN

Suppose we are interested in using the BBN shown in Figure 5.13 to diagnose whether a person has heart disease. The following cases illustrate how the diagnosis can be made under different scenarios.

Case 1: No Prior Information

Without any prior information, we can determine whether the person is likely to have heart disease by computing the prior probabilities $P(\texttt{HD} = \texttt{Yes})$ and $P(\texttt{HD} = \texttt{No})$. To simplify the notation, let $\alpha \in \{\texttt{Yes}, \texttt{No}\}$ denote the binary values of `Exercise` and $\beta \in \{\texttt{Healthy}, \texttt{Unhealthy}\}$ denote the binary values of `Diet`.

$$\begin{aligned}
P(\texttt{HD} = \texttt{Yes}) &= \sum_\alpha \sum_\beta P(\texttt{HD} = \texttt{Yes}|E = \alpha, D = \beta) P(E = \alpha, D = \beta) \\
&= \sum_\alpha \sum_\beta P(\texttt{HD} = \texttt{Yes}|E = \alpha, D = \beta) P(E = \alpha) P(D = \beta) \\
&= 0.25 \times 0.7 \times 0.25 + 0.45 \times 0.7 \times 0.75 + 0.55 \times 0.3 \times 0.25 \\
&\quad + 0.75 \times 0.3 \times 0.75 \\
&= 0.49.
\end{aligned}$$

Since $P(\texttt{HD} = \texttt{no}) = 1 - P(\texttt{HD} = \texttt{yes}) = 0.51$, the person has a slightly higher chance of not getting the disease.

Case 2: High Blood Pressure

If the person has high blood pressure, we can make a diagnosis about heart disease by comparing the posterior probabilities, $P(\texttt{HD} = \texttt{Yes}|\texttt{BP} = \texttt{High})$ against $P(\texttt{HD} = \texttt{No}|\texttt{BP} = \texttt{High})$. To do this, we must compute $P(\texttt{BP} = \texttt{High})$:

$$\begin{aligned} P(\texttt{BP} = \texttt{High}) &= \sum_{\gamma} P(\texttt{BP} = \texttt{High}|\texttt{HD} = \gamma)P(\texttt{HD} = \gamma) \\ &= 0.85 \times 0.49 + 0.2 \times 0.51 = 0.5185. \end{aligned}$$

where $\gamma \in \{\texttt{Yes}, \texttt{No}\}$. Therefore, the posterior probability the person has heart disease is

$$\begin{aligned} P(\texttt{HD} = \texttt{Yes}|\texttt{BP} = \texttt{High}) &= \frac{P(\texttt{BP} = \texttt{High}|\texttt{HD} = \texttt{Yes})P(\texttt{HD} = \texttt{Yes})}{P(\texttt{BP} = \texttt{High})} \\ &= \frac{0.85 \times 0.49}{0.5185} = 0.8033. \end{aligned}$$

Similarly, $P(\texttt{HD} = \texttt{No}|\texttt{BP} = \texttt{High}) = 1 - 0.8033 = 0.1967$. Therefore, when a person has high blood pressure, it increases the risk of heart disease.

Case 3: High Blood Pressure, Healthy Diet, and Regular Exercise

Suppose we are told that the person exercises regularly and eats a healthy diet. How does the new information affect our diagnosis? With the new information, the posterior probability that the person has heart disease is

$$\begin{aligned} & P(\texttt{HD} = \texttt{Yes}|\texttt{BP} = \texttt{High}, D = \texttt{Healthy}, E = \texttt{Yes}) \\ = & \left[\frac{P(\texttt{BP} = \texttt{High}|\texttt{HD} = \texttt{Yes}, D = \texttt{Healthy}, E = \texttt{Yes})}{P(\texttt{BP} = \texttt{High}|D = \texttt{Healthy}, E = \texttt{Yes})}\right] \\ & \quad \times P(\texttt{HD} = \texttt{Yes}|D = \texttt{Healthy}, E = \texttt{Yes}) \\ = & \frac{P(\texttt{BP} = \texttt{High}|\texttt{HD} = \texttt{Yes})P(\texttt{HD} = \texttt{Yes}|D = \texttt{Healthy}, E = \texttt{Yes})}{\sum_{\gamma} P(\texttt{BP} = \texttt{High}|\texttt{HD} = \gamma)P(\texttt{HD} = \gamma|D = \texttt{Healthy}, E = \texttt{Yes})} \\ = & \frac{0.85 \times 0.25}{0.85 \times 0.25 + 0.2 \times 0.75} \\ = & \ 0.5862, \end{aligned}$$

while the probability that the person does not have heart disease is

$$P(\texttt{HD} = \texttt{No}|\texttt{BP} = \texttt{High}, D = \texttt{Healthy}, E = \texttt{Yes}) = 1 - 0.5862 = 0.4138.$$

The model therefore suggests that eating healthily and exercising regularly may reduce a person's risk of getting heart disease.

Characteristics of BBN

Following are some of the general characteristics of the BBN method:

1. BBN provides an approach for capturing the prior knowledge of a particular domain using a graphical model. The network can also be used to encode causal dependencies among variables.

2. Constructing the network can be time consuming and requires a large amount of effort. However, once the structure of the network has been determined, adding a new variable is quite straightforward.

3. Bayesian networks are well suited to dealing with incomplete data. Instances with missing attributes can be handled by summing or integrating the probabilities over all possible values of the attribute.

4. Because the data is combined probabilistically with prior knowledge, the method is quite robust to model overfitting.

5.4 Artificial Neural Network (ANN)

The study of artificial neural networks (ANN) was inspired by attempts to simulate biological neural systems. The human brain consists primarily of nerve cells called **neurons**, linked together with other neurons via strands of fiber called **axons**. Axons are used to transmit nerve impulses from one neuron to another whenever the neurons are stimulated. A neuron is connected to the axons of other neurons via **dendrites**, which are extensions from the cell body of the neuron. The contact point between a dendrite and an axon is called a **synapse**. Neurologists have discovered that the human brain learns by changing the strength of the synaptic connection between neurons upon repeated stimulation by the same impulse.

Analogous to human brain structure, an ANN is composed of an interconnected assembly of nodes and directed links. In this section, we will examine a family of ANN models, starting with the simplest model called **perceptron**, and show how the models can be trained to solve classification problems.

Optimizations.
Genetic Algorithm.
Genetic Programming.

From:

Programming Collective Intelligence

by Toby Segaran

CHAPTER 5
Optimization

This chapter will look at how to solve collaboration problems using a set of techniques called *stochastic optimization*. Optimization techniques are typically used in problems that have many possible solutions across many variables, and that have outcomes that can change greatly depending on the combinations of these variables. These optimization techniques have a wide variety of applications: we use them in physics to study molecular dynamics, in biology to predict protein structures, and in computer science to determine the worst possible running time of an algorithm. NASA even uses optimization techniques to design antennas that have the right operating characteristics, which look unlike anything a human designer would create.

Optimization finds the best solution to a problem by trying many different solutions and scoring them to determine their quality. Optimization is typically used in cases where there are too many possible solutions to try them all. The simplest but least effective method of searching for solutions is just trying a few thousand random guesses and seeing which one is best. More effective methods, which will be discussed in this chapter, involve intelligently modifying the solutions in a way that is likely to improve them.

The first example in this chapter concerns group travel planning. Anyone who has planned a trip for a group of people, or perhaps even for an individual, realizes that there are a lot of different inputs required, such as what everyone's flight schedule should be, how many cars should be rented, and which airport is easiest. Many outputs must also be considered, such as total cost, time spent waiting at airports, and time taken off work. Because the inputs can't be mapped to the outputs with a simple formula, the problem of finding the best solution lends itself to optimization.

The other examples in the chapter show the flexibility of optimization by considering two completely different problems: how to allocate limited resources based on people's preferences, and how to visualize a social network with minimal crossed lines. By the end of the chapter, you'll be able to spot other types of problems that can be solved using optimization.

Group Travel

Planning a trip for a group of people (the Glass family in this example) from different locations all arriving at the same place is always a challenge, and it makes for an interesting optimization problem. To begin, create a new file called *optimization.py* and insert the following code:

```
import time
import random
import math

people = [('Seymour','BOS'),
          ('Franny','DAL'),
          ('Zooey','CAK'),
          ('Walt','MIA'),
          ('Buddy','ORD'),
          ('Les','OMA')]

# LaGuardia airport in New York
destination='LGA'
```

The family members are from all over the country and wish to meet up in New York. They will all arrive on the same day and leave on the same day, and they would like to share transportation to and from the airport. There are dozens of flights per day to New York from any of the family members' locations, all leaving at different times. The flights also vary in price and in duration.

You can download a sample file of flight data from *http://kiwitobes.com/optimize/schedule.txt*.

This file contains origin, destination, departure time, arrival time, and price for a set of flights in a comma-separated format:

```
LGA,MIA,20:27,23:42,169
MIA,LGA,19:53,22:21,173
LGA,BOS,6:39,8:09,86
BOS,LGA,6:17,8:26,89
LGA,BOS,8:23,10:28,149
```

Load this data into a dictionary with the `origin` and destination (`dest`) as the keys and a list of potential flight details as the values. Add this code to load the data into *optimization.py*:

```
flights={}
#
for line in file('schedule.txt'):
  origin,dest,depart,arrive,price=line.strip( ).split(',')
  flights.setdefault((origin,dest),[])

  # Add details to the list of possible flights
  flights[(origin,dest)].append((depart,arrive,int(price)))
```

It's also useful at this point to define a utility function, getminutes, which calculates how many minutes into the day a given time is. This makes it easy to calculate flight times and waiting times. Add this function to *optimization.py*:

```
def getminutes(t):
  x=time.strptime(t,'%H:%M')
  return x[3]*60+x[4]
```

The challenge now is to decide which flight each person in the family should take. Of course, keeping total price down is a goal, but there are many other possible factors that the optimal solution will take into account and try to minimize, such as total waiting time at the airport or total flight time. These other factors will be discussed in more detail shortly.

Representing Solutions

When approaching a problem like this, it's necessary to determine how a potential solution will be represented. The optimization functions you'll see later are generic enough to work on many different types of problems, so it's important to choose a simple representation that's not specific to the group travel problem. A very common representation is a list of numbers. In this case, each number can represent which flight a person chooses to take, where 0 is the first flight of the day, 1 is the second, and so on. Since each person needs an outbound flight and a return flight, the length of this list is twice the number of people.

For example, the list:

```
[1,4,3,2,7,3,6,3,2,4,5,3]
```

Represents a solution in which Seymour takes the second flight of the day from Boston to New York, and the fifth flight back to Boston on the day he returns. Franny takes the fourth flight from Dallas to New York, and the third flight back.

Because it will be difficult to interpret solutions from this list of numbers, you'll need a routine that prints all the flights that people decide to take in a nice table. Add this function to *optimization.py*:

```
def printschedule(r):
  for d in range(len(r)/2):
    name=people[d][0]
    origin=people[d][1]
    out=flights[(origin,destination)][r[d]]
    ret=flights[(destination,origin)][r[d+1]]
    print '%10s%10s %5s-%5s $%3s %5s-%5s $%3s' % (name,origin,
                                                  out[0],out[1],out[2],
                                                  ret[0],ret[1],ret[2])
```

This will print a line containing each person's name and origin, as well as the departure time, arrival time, and price for the outgoing and return flights. Try this function in your Python session:

```
>>> import optimization
>>> s=[1,4,3,2,7,3,6,3,2,4,5,3]
>>> optimization.printschedule(s)
   Seymour    Boston 12:34-15:02 $109 12:08-14:05 $142
    Franny    Dallas 12:19-15:25 $342  9:49-13:51 $229
     Zooey     Akron  9:15-12:14 $247 15:50-18:45 $243
      Walt     Miami 15:34-18:11 $326 14:08-16:09 $232
     Buddy   Chicago 14:22-16:32 $126 15:04-17:23 $189
       Les     Omaha 15:03-16:42 $135  6:19- 8:13 $239
```

Even disregarding price, this schedule has some problems. In particular, since the family members are traveling to and from the airport together, everyone has to arrive at the airport at 6 a.m. for Les's return flight, even though some of them don't leave until nearly 4 p.m. To determine the best combination, the program needs a way of weighting the various properties of different schedules and deciding which is the best.

The Cost Function

The *cost function* is the key to solving any problem using optimization, and it's usually the most difficult thing to determine. The goal of any optimization algorithm is to find a set of inputs—flights, in this case—that minimizes the cost function, so the cost function has to return a value that represents how bad a solution is. There is no particular scale for *badness*; the only requirement is that the function returns larger values for worse solutions.

Often it is difficult to determine what makes a solution good or bad across many variables. Consider a few of the things that can be measured in the group travel example:

Price
The total price of all the plane tickets, or possibly a weighted average that takes financial situations into account.

Travel time
The total time that everyone has to spend on a plane.

Waiting time
Time spent at the airport waiting for the other members of the party to arrive.

Departure time
Flights that leave too early in the morning may impose an additional cost by requiring travelers to miss out on sleep.

Car rental period
If the party rents a car, they must return it earlier in the day than when they rented it, or be forced to pay for a whole extra day.

It's not too hard to think of even more aspects of a particular schedule that could make the experience more or less pleasant. Any time you're faced with finding the best solution to a complicated problem, you'll need to decide what the important factors are. Although this can be difficult, the big advantage is that once it's done, you can use the optimization algorithms in this chapter on almost any problem with minimal modification.

After choosing some variables that impose costs, you'll need to determine how to combine them into a single number. In this example, it's necessary to decide, for instance, how much money that time on the plane or time waiting in the airport is worth. You might decide that it's worth spending $1 for every minute saved on air travel (this translates into spending an extra $90 for a direct flight that saves an hour and a half), and $0.50 for every minute saved waiting in the airport. You could also add the cost of an extra day of car rental if everyone returns to the airport at a later time of the day than when they first rented the car.

There are a huge number of possibilities for the `getcost` function defined here. This function takes into account the total cost of the trip and the total time spent waiting at airports for the various members of the family. It also adds a penalty of $50 if the car is returned at a later time of the day than when it was rented. Add this function to *optimization.py*, and feel free to add additional costs or to tweak the relative importance of money and time:

```
def schedulecost(sol):
  totalprice=0
  latestarrival=0
  earliestdep=24*60

  for d in range(len(sol)/2):
    # Get the inbound and outbound flights
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[d])]
    returnf=flights[(destination,origin)][int(sol[d+1])]

    # Total price is the price of all outbound and return flights
    totalprice+=outbound[2]
    totalprice+=returnf[2]

    # Track the latest arrival and earliest departure
    if latestarrival<getminutes(outbound[1]): latestarrival=getminutes(outbound[1])
    if earliestdep>getminutes(returnf[0]): earliestdep=getminutes(returnf[0])

  # Every person must wait at the airport until the latest person arrives.
  # They also must arrive at the same time and wait for their flights.
  totalwait=0
  for d in range(len(sol)/2):
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[d])]
    returnf=flights[(destination,origin)][int(sol[d+1])]
    totalwait+=latestarrival-getminutes(outbound[1])
    totalwait+=getminutes(returnf[0])-earliestdep
  # Does this solution require an extra day of car rental? That'll be $50!
  if latestarrival>earliestdep: totalprice+=50

  return totalprice+totalwait
```

The logic in this function is quite simplistic, but it illustrates the point. It can be enhanced in several ways—right now, the total wait time assumes that all the family members will leave the airport together when the last person arrives, and will all go to the airport for the earliest departure. This can be modified so that anyone facing a two-hour or longer wait rents his own car instead, and the prices and waiting time can be adjusted accordingly.

You can try this function in your Python session:

```
>>> reload(optimization)
>>> optimization.schedulecost(s)
5285
```

Now that the cost function has been created, it should be clear that the goal is to minimize cost by choosing the correct set of numbers. In theory, you could try every possible combination, but in this example there are 16 flights, all with 9 possibilities, giving a total of 9^{16} (around 300 billion) combinations. Testing every combination would guarantee you'd get the best answer, but it would take a very long time on most computers.

Random Searching

Random searching isn't a very good optimization method, but it makes it easy to understand exactly what all the algorithms are trying to do, and it also serves as a baseline so you can see if the other algorithms are doing a good job.

The function takes a couple of parameters. Domain is a list of 2-tuples that specify the minimum and maximum values for each variable. The length of the solution is the same as the length of this list. In the current example, there are nine outbound flights and nine inbound flights for every person, so the domain in the list is (0,8) repeated twice for each person.

The second parameter, costf, is the cost function, which in this example will be schedulecost. This is passed as a parameter so that the function can be reused for other optimization problems. This function randomly generates 1,000 guesses and calls costf on them. It keeps track of the best guess (the one with the lowest cost) and returns it. Add it to *optimization.py*:

```
def randomoptimize(domain,costf):
  best=999999999
  bestr=None
  for i in range(1000):
    # Create a random solution
    r=[random.randint(domain[i][0],domain[i][1])
       for i in range(len(domain))]
```

```
        # Get the cost
        cost=costf(r)

        # Compare it to the best one so far
        if cost<best:
          best=cost
          bestr=r
      return r
```

Of course, 1,000 guesses is a very small fraction of the total number of possibilities. However, this example has many possibilities that are good (if not the best), so with a thousand tries, the function will likely come across a solution that isn't awful. Try it in your Python session:

```
>>> reload(optimization)
>>> domain=[(0,8)]*(len(optimization.people)*2)
>>> s=optimization.randomoptimize(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
3328
>>> optimization.printschedule(s)
   Seymour    Boston 12:34-15:02 $109 12:08-14:05 $142
    Franny    Dallas 12:19-15:25 $342  9:49-13:51 $229
     Zooey     Akron  9:15-12:14 $247 15:50-18:45 $243
      Walt     Miami 15:34-18:11 $326 14:08-16:09 $232
     Buddy   Chicago 14:22-16:32 $126 15:04-17:23 $189
       Les     Omaha 15:03-16:42 $135  6:19- 8:13 $239
```

Due to the random element, your results will be different from the results here. The results shown are not great, as they have Zooey waiting at the airport for six hours until Walt arrives, but they could definitely be worse. Try running this function several times to see if the cost changes very much, or try increasing the loop size to 10,000 to see if you find better results that way.

Hill Climbing

Randomly trying different solutions is very inefficient because it does not take advantage of the good solutions that have already been discovered. In our example, a schedule with a low overall cost is probably similar to other schedules that have a low cost. Because random optimization jumps around, it won't automatically look at similar schedules to locate the good ones that have already been found.

An alternate method of random searching is called *hill climbing*. Hill climbing starts with a random solution and looks at the set of neighboring solutions for those that are better (have a lower cost function). This is analogous to going down a hill, as shown in Figure 5-1.

Imagine you are the person shown in the figure, having been randomly dropped into this landscape. You want to reach the lowest point to find water. To do this, you might look in each direction and walk toward wherever the land slopes downward

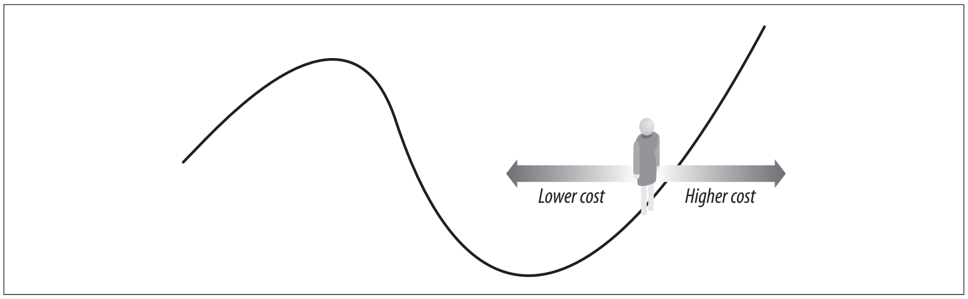


Figure 5-1. Seeking the lowest cost on a hill

most steeply. You would continue to walk in the most steeply sloping direction until you reached a point where the terrain was flat or began sloping uphill.

You can apply this hill climbing approach to the task of finding the best travel schedule for the Glass family. Start with a random schedule and find all the neighboring schedules. In this case, that means finding all the schedules that have one person on a slightly earlier or slightly later flight. The cost is calculated for each of the neighboring schedules, and the one with the lowest cost becomes the new solution. This process is repeated until none of the neighboring schedules improves the cost.

To implement this, add `hillclimb` to *optimization.py*:

```
def hillclimb(domain,costf):
  # Create a random solution
  sol=[random.randint(domain[i][0],domain[i][1])
     for i in range(len(domain))]

  # Main loop
  while 1:

    # Create list of neighboring solutions
    neighbors=[]
    for j in range(len(domain)):

      # One away in each direction
      if sol[j]>domain[j][0]:
        neighbors.append(sol[0:j]+[sol[j]+1]+sol[j+1:])
      if sol[j]<domain[j][1]:
        neighbors.append(sol[0:j]+[sol[j]-1]+sol[j+1:])

    # See what the best solution amongst the neighbors is
    current=costf(sol)
    best=current
    for j in range(len(neighbors)):
      cost=costf(neighbors[j])
      if cost<best:
        best=cost
        sol=neighbors[j]
    # If there's no improvement, then we've reached the top
    if best==current:
      break

  return sol
```

This function generates a random list of numbers within the given domain to create the initial solution. It finds all the neighbors for the current solution by looping over every element in the list and then creating two new lists with that element increased by one and decreased by one. The best of these neighbors becomes the new solution.

Try this function in your Python session to see how it compares to randomly searching for a solution:

```
>>> s=optimization.hillclimb(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
3063
>>> optimization.printschedule(s)
   Seymour       BOS 12:34-15:02 $109 10:33-12:03 $ 74
    Franny       DAL 10:30-14:57 $290 10:51-14:16 $256
     Zooey       CAK 10:53-13:36 $189 10:32-13:16 $139
      Walt       MIA 11:28-14:40 $248 12:37-15:05 $170
     Buddy       ORD 12:44-14:17 $134 10:33-13:11 $132
       Les       OMA 11:08-13:07 $175 18:25-20:34 $205
```

This function runs quickly and usually finds a better solution than randomly searching. There is, however, one major drawback to hill climbing. Look at Figure 5-2.

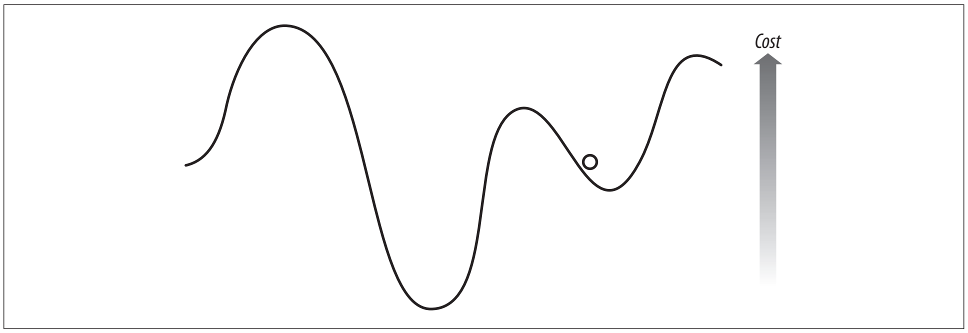


Figure 5-2. Stuck in a local minimum

From this figure it's clear that simply moving down the slope will not necessarily lead to the best solution overall. The final solution will be a *local minimum*, a solution better than those around it but not the best overall. The best overall is called the *global minimum*, which is what optimization algorithms are ultimately supposed to find. One approach to this dilemma is called *random-restart hill climbing*, where the hill climbing algorithm is run several times with random starting points in the hope that one of them will be close to the global minimum. The next two sections, "Simulated Annealing" and "Genetic Algorithms," show other ways to avoid getting stuck in a local minimum.

Simulated Annealing

Simulated annealing is an optimization method inspired by physics. Annealing is the process of heating up an alloy and then cooling it down slowly. Because the atoms are first made to jump around a lot and then gradually settle into a low energy state, the atoms can find a low energy configuration.

The algorithm version of annealing begins with a random solution to the problem. It uses a variable representing the temperature, which starts very high and gradually gets lower. In each iteration, one of the numbers in the solution is randomly chosen and changed in a certain direction. In our example, Seymour's return flight might be moved from the second of the day to the third. The cost is calculated before and after the change, and the costs are compared.

Here's the important part: if the new cost is lower, the new solution becomes the current solution, which is very much like the hill-climbing method. However, if the cost is *higher*, the new solution can still become the current solution with a certain probability. This is an attempt to avoid the local minimum problem shown in Figure 5-2.

In some cases, it's necessary to move to a worse solution before you can get to a better one. Simulated annealing works because it will always accept a move for the better, and because it is willing to accept a worse solution near the beginning of the process. As the process goes on, the algorithm becomes less and less likely to accept a worse solution, until at the end it will only accept a better solution. The probability of a higher-cost solution being accepted is given by this formula:

$$p=e^{((\text{-highcost–lowcost})/\text{temperature})}$$

Since the temperature (the willingness to accept a worse solution) starts very high, the exponent will always be close to 0, so the probability will almost be 1. As the temperature decreases, the difference between the high cost and the low cost becomes more important—a bigger difference leads to a lower probability, so the algorithm will favor only slightly worse solutions over much worse ones.

Create a new function in *optimization.py* called `annealingoptimize`, which implements this algorithm:

```
def annealingoptimize(domain,costf,T=10000.0,cool=0.95,step=1):
  # Initialize the values randomly
  vec=[float(random.randint(domain[i][0],domain[i][1]))
       for i in range(len(domain))]

  while T>0.1:
    # Choose one of the indices
    i=random.randint(0,len(domain)-1)

    # Choose a direction to change it
    dir=random.randint(-step,step)
    # Create a new list with one of the values changed
    vecb=vec[:]
    vecb[i]+=dir
    if vecb[i]<domain[i][0]: vecb[i]=domain[i][0]
    elif vecb[i]>domain[i][1]: vecb[i]=domain[i][1]

    # Calculate the current cost and the new cost
    ea=costf(vec)
    eb=costf(vecb)
    p=pow(math.e,(-eb-ea)/T)

    # Is it better, or does it make the probability
    # cutoff?
    if (eb<ea or random.random( )<p):
      vec=vecb

    # Decrease the temperature
    T=T*cool
  return vec
```

To do annealing, this function first creates a random solution of the right length with all the values in the range specified by the domain parameter. The temperature and the cooling rate are optional parameters. In each iteration, i is set to a random index of the solution, and dir is set to a random number between –step and step. It calculates the current function cost and the cost if it were to change the value at i by dir.

The line of code in bold shows the probability calculation, which gets lower as T gets lower. If a random float between 0 and 1 is less than this value, or if the new solution is better, the function accepts the new solution. The function loops until the temperature has almost reached 0, each time multiplying it by the cooling rate.

Now you can try to optimize with simulated annealing in your Python session:

```
>>> reload(optimization)
>>> s=optimization.annealingoptimize(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
2278
>>> optimization.printschedule(s)
   Seymour    Boston 12:34-15:02 $109 10:33-12:03 $ 74
    Franny    Dallas 10:30-14:57 $290 10:51-14:16 $256
     Zooey     Akron 10:53-13:36 $189 10:32-13:16 $139
      Walt     Miami 11:28-14:40 $248 12:37-15:05 $170
     Buddy   Chicago 12:44-14:17 $134 10:33-13:11 $132
       Les     Omaha 11:08-13:07 $175 15:07-17:21 $129
```

This optimization did a good job of reducing the overall wait times while keeping the costs down. Obviously, your results will be different, and there is a chance that they will be worse. For any given problem, it's a good idea to experiment with different parameters for the initial temperature and the cooling rate. You can also vary the possible step size for the random movements.

Genetic Algorithms

Another set of techniques for optimization, also inspired by nature, is called *genetic algorithms*. These work by initially creating a set of random solutions known as the *population*. At each step of the optimization, the cost function for the entire population is calculated to get a ranked list of solutions. An example is shown in Table 5-1.

Table 5-1. Ranked list of solutions and costs

Solution	Cost
[7, 5, 2, 3, 1, 6, 1, 6, 7, 1, 0, 3]	4394
[7, 2, 2, 2, 3, 3, 2, 3, 5, 2, 0, 8]	4661
…	…
[0, 4, 0, 3, 8, 8, 4, 4, 8, 5, 6, 1]	7845
[5, 8, 0, 2, 8, 8, 8, 2, 1, 6, 6, 8]	8088

After the solutions are ranked, a new population—known as the next *generation*—is created. First, the top solutions in the current population are added to the new population as they are. This process is called *elitism*. The rest of the new population consists of completely new solutions that are created by modifying the best solutions.

There are two ways that solutions can be modified. The simpler of these is called *mutation*, which is usually a small, simple, random change to an existing solution. In this case, a mutation can be done simply by picking one of the numbers in the solution and increasing or decreasing it. A couple of examples are shown in Figure 5-3.

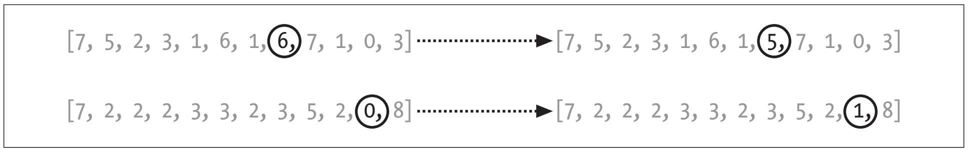


Figure 5-3. Examples of mutating a solution

The other way to modify solutions is called *crossover* or *breeding*. This method involves taking two of the best solutions and combining them in some way. In this case, a simple way to do crossover is to take a random number of elements from one solution and the rest of the elements from another solution, as illustrated in Figure 5-4.

A new population, usually the same size as the old one, is created by randomly mutating and breeding the best solutions. Then the process repeats—the new population is ranked and another population is created. This continues either for a fixed number of iterations or until there has been no improvement over several generations.

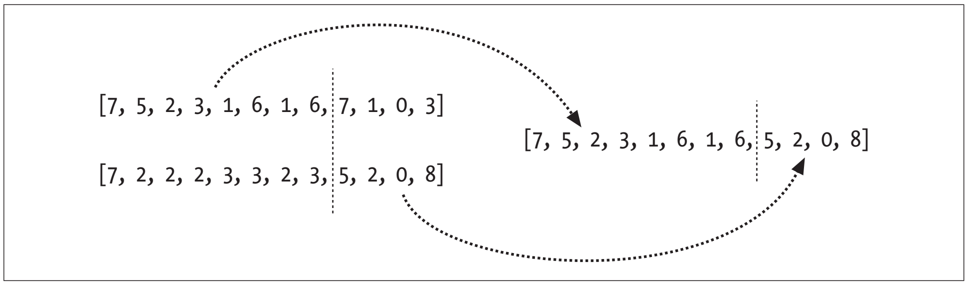


Figure 5-4. Example of crossover

Add geneticoptimize to *optimization.py*:

```
def geneticoptimize(domain,costf,popsize=50,step=1,
                    mutprod=0.2,elite=0.2,maxiter=100):
  # Mutation Operation
  def mutate(vec):
    i=random.randint(0,len(domain)-1)
    if random.random( )<0.5 and vec[i]>domain[i][0]:
      return vec[0:i]+[vec[i]-step]+vec[i+1:]
    elif vec[i]<domain[i][1]:
      return vec[0:i]+[vec[i]+step]+vec[i+1:]

  # Crossover Operation
  def crossover(r1,r2):
    i=random.randint(1,len(domain)-2)
    return r1[0:i]+r2[i:]

  # Build the initial population
  pop=[]
  for i in range(popsize):
    vec=[random.randint(domain[i][0],domain[i][1])
         for i in range(len(domain))]
    pop.append(vec)

  # How many winners from each generation?
  topelite=int(elite*popsize)

  # Main loop
  for i in range(maxiter):
    scores=[(costf(v),v) for v in pop]
    scores.sort( )
    ranked=[v for (s,v) in scores]

    # Start with the pure winners
    pop=ranked[0:topelite]

    # Add mutated and bred forms of the winners
    while len(pop)<popsize:
      if random.random( )<mutprob:
        # Mutation
        c=random.randint(0,topelite)
        pop.append(mutate(ranked[c]))
      else:

        # Crossover
        c1=random.randint(0,topelite)
        c2=random.randint(0,topelite)
        pop.append(crossover(ranked[c1],ranked[c2]))

    # Print current best score
    print scores[0][0]

  return scores[0][1]
```

This function takes several optional parameters:

`popsize`
: The size of the population

`mutprob`
: The probability that a new member of the population will be a mutation rather than a crossover

`elite`
: The fraction of the population that are considered good solutions and are allowed to pass into the next generation

`maxiter`
: The number of generations to run

Try optimizing the travel plans using the genetic algorithm in your Python session:

```
>>> s=optimization.geneticoptimize(domain,optimization.schedulecost)
3532
3503
...
2591
2591
2591
>>> optimization.printschedule(s)
   Seymour       BOS 12:34-15:02 $109 10:33-12:03 $ 74
    Franny       DAL 10:30-14:57 $290 10:51-14:16 $256
     Zooey       CAK 10:53-13:36 $189 10:32-13:16 $139
      Walt       MIA 11:28-14:40 $248 12:37-15:05 $170
     Buddy       ORD 12:44-14:17 $134 10:33-13:11 $132
       Les       OMA 11:08-13:07 $175 11:07-13:24 $171
```

In Chapter 11, you'll see an extension of genetic algorithms called *genetic programming*, where similar ideas are used to create entirely new programs.

The computer scientist John Holland is widely considered to be the father of genetic algorithms because of his 1975 book, *Adaptation in Natural and Artificial Systems* (University of Michigan Press). Yet the work goes back to biologists in the 1950s who were attempting to model evolution on computers. Since then, genetic algorithms and other optimization methods have been used for a huge variety of problems, including:

- Finding which concert hall shape gives the best acoustics
- Designing an optimal wing for a supersonic aircraft
- Suggesting the best library of chemicals to research as potential drugs
- Automatically designing a chip for voice recognition

Potential solutions to these problems can be turned into lists of numbers. This makes it easy to apply genetic algorithms or simulated annealing.

Whether a particular optimization method will work depends very much on the problem. Simulated annealing, genetic optimization, and most other optimization methods rely on the fact that, in most problems, the best solution is close to other good solutions. To see a case where optimization might not work, look at Figure 5-5.

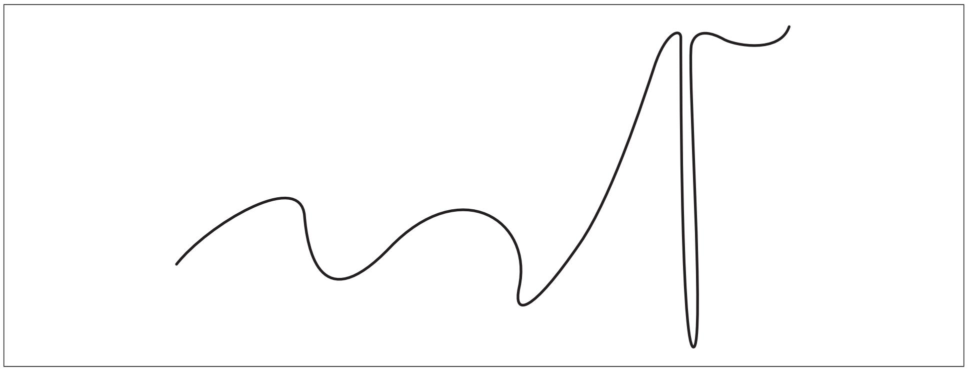


Figure 5-5. Poor problem for optimization

The cost is actually lowest at a very steep point on the far right of the figure. Any solution that is close by would probably be dismissed from consideration because of its high cost, and you would never find your way to the global minimum. Most algorithms would settle in one of the local minima on the left side of the figure.

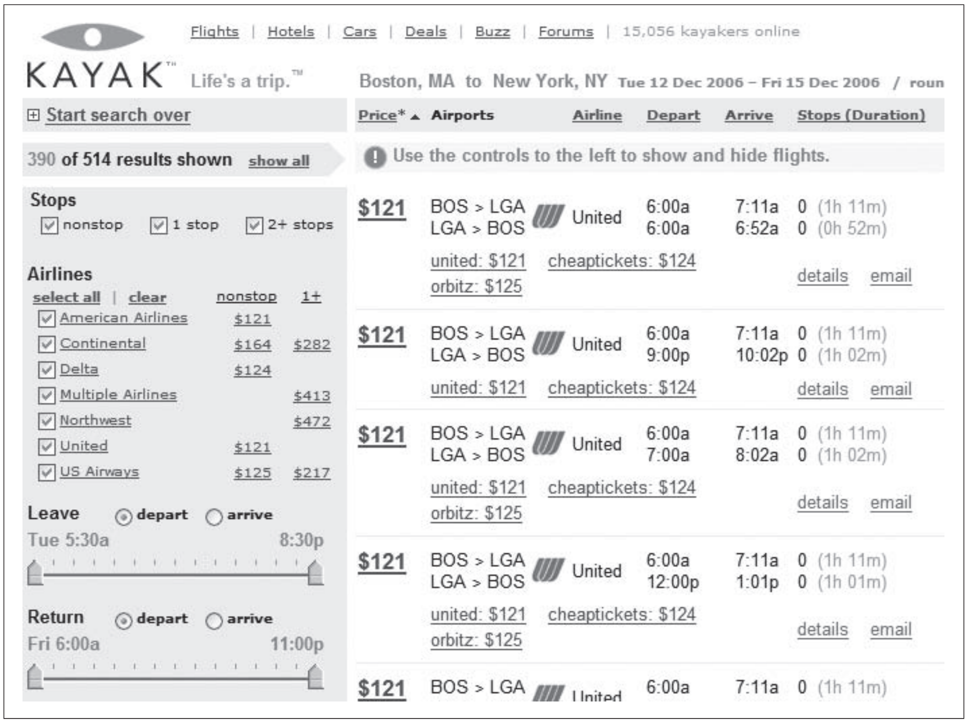
The flight scheduling example works because moving a person from the second to the third flight of the day would probably change the overall cost by a smaller amount than moving that person to the eighth flight of the day would. If the flights were in random order, the optimization methods would work no better than a random search—in fact, there's no optimization method that will consistently work better than a random search in that case.

Real Flight Searches

Now that everything is working with the sample data, it's time to try getting real flight data to see if the same optimizations can be used. You'll be downloading data from Kayak, which provides an API for doing flight searches. The main difference between real flight data and the sample you've been working with is that in the real flight data, there are many more than nine flights per day between most major cities.

The Kayak API

Kayak, shown in Figure 5-6, is a popular *vertical search engine* for travel. Although there are lots of travel sites online, Kayak is useful for this example because it has a nice XML API that can be used to perform real travel searches from within a Python program. To use the API, you'll need to sign up for a developer key by going to *http://www.kayak.com/labs/api/search*.


Flights | Hotels | Cars | Deals | Buzz | Forums | 15,056 kayakers online
KAYAK™ Life's a trip.™
⊞ Start search over
show all
Use the controls to the left to show and hide flights.
details
email

Boston, MA to New York, NY Tue 12 Dec 2006 – Fri 15 Dec 2006 / roun

390 of 514 results shown

Stops ☑ nonstop ☑ 1 stop ☑ 2+ stops

Airlines select all | clear

	nonstop	1+
☑ American Airlines	$121	
☑ Continental	$164	$282
☑ Delta	$124	
☑ Multiple Airlines		$413
☑ Northwest		$472
☑ United	$121	
☑ US Airways	$125	$217

Leave ● depart ○ arrive
Tue 5:30a 8:30p

Return ● depart ○ arrive
Fri 6:00a 11:00p

Price* ▲	Airports	Airline	Depart	Arrive	Stops (Duration)
$121	BOS > LGA LGA > BOS	United	6:00a 6:00a	7:11a 6:52a	0 (1h 11m) 0 (0h 52m)
	united: $121 cheaptickets: $124 orbitz: $125				
$121	BOS > LGA LGA > BOS	United	6:00a 9:00p	7:11a 10:02p	0 (1h 11m) 0 (1h 02m)
	united: $121 cheaptickets: $124				
$121	BOS > LGA LGA > BOS	United	6:00a 7:00a	7:11a 8:02a	0 (1h 11m) 0 (1h 02m)
	united: $121 cheaptickets: $124 orbitz: $125				
$121	BOS > LGA LGA > BOS	United	6:00a 12:00p	7:11a 1:01p	0 (1h 11m) 0 (1h 01m)
	united: $121 cheaptickets: $124 orbitz: $125				
$121	BOS > LGA	United	6:00a	7:11a	0 (1h 11m)

Figure 5-6. Screenshot of the Kayak travel search interface

The developer key is a long string of numbers and letters that you'll use to do flight searches in Kayak (it can also be used for hotel searches, but that won't be covered here). At the time of writing, there is not a specific Python API for Kayak like there is for del.icio.us, but the XML interface is very well explained. This chapter will show you how to create searches using the Python packages *urllib2* and *xml.dom.minidom*, both of which are included with the standard Python distribution.

The minidom Package

The *minidom* package is part of the standard Python distribution. It is a lightweight implementation of the Document Object Model (DOM) interface, a standard way of treating an XML document as a tree of objects. The package takes strings or open files containing XML and returns an object that you can use to easily extract information. For example, enter the following in a Python session:

```
>>> import xml.dom.minidom
>>> dom=xml.dom.minidom.parseString('<data><rec>Hello!</rec></data>')
>>> dom
<xml.dom.minidom.Document instance at 0x00980C38>
>>> r=dom.getElementsByTagName('rec')
>>> r
[<DOM Element: rec at 0xa42350>]
>>> r[0].firstChild
<DOM Text node "Hello!">
>>> r[0].firstChild.data
u'Hello!'
```

Because many web sites now offer a way to access information through an XML interface, learning how to use the Python XML packages is very useful for collective intelligence programming. Here are the important methods of DOM objects that you'll be using for the Kayak API:

`getElementsByTagName(name)`
: Returns a list of all DOM nodes by searching throughout the whole document for elements whose tag matches `name`.

`firstChild`
: Returns the first child node of this object. In the above example, the first child of `r` is the node representing the text "Hello."

`data`
: Returns the data associated with this object, which in most cases is a Unicode string of the text that the node contains.

Flight Searches

Begin by creating a new file called *kayak.py* and adding the following statements:

```
import time
import urllib2
import xml.dom.minidom

kayakkey='YOURKEYHERE'
```

The first thing you'll need is code to get a new Kayak session using your developer key. The function to do this sends a request to apisession with the token parameter set to your developer key. The XML returned by this URL will contain a tag sid, with a session ID inside it:

```
<sid>1-hX4lII_wS$8b06a07kHj</sid>
```

The function just has to parse the XML to extract the contents of the sid tag. Add this function to *kayak.py*:

```
def getkayaksession( ):
  # Construct the URL to start a session
  url='http://www.kayak.com/k/ident/apisession?token=%s&version=1' % kayakkey

  # Parse the resulting XML
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))

  # Find <sid>xxxxxxxx</sid>
  sid=doc.getElementsByTagName('sid')[0].firstChild.data
  return sid
```

The next step is to create a function to start the flight search. The URL for this search is very long because it contains all the parameters for the flight search. The important parameters for this search are sid (the session ID returned by getkayaksession), destination, and depart_date.

The resulting XML has a tag called searchid, which the function will extract in the same manner as getkayaksession. Since the search may take a long time, this call doesn't actually return any results—it just begins the search and returns an ID that can be used to poll for the results.

Add this function to *kayak.py*:

```
def flightsearch(sid,origin,destination,depart_date):

  # Construct search URL
  url='http://www.kayak.com/s/apisearch?basicmode=true&oneway=y&origin=%s' % origin
  url+='&destination=%s&depart_date=%s' % (destination,depart_date)
  url+='&return_date=none&depart_time=a&return_time=a'
  url+='&travelers=1&cabin=e&action=doFlights&apimode=1'
  url+='&_sid_=%s&version=1' % (sid)

  # Get the XML
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))

  # Extract the search ID
  searchid=doc.getElementsByTagName('searchid')[0].firstChild.data

  return searchid
```

Finally, you'll need a function that requests the results until there are no more. Kayak provides another URL, `flight`, which gives these results. In the returned XML, there is a tag called `morepending`, which contains the word "true" until the search is complete. The function has to request the page until `morepending` is no longer true, and then the functions gets the complete results.

Add this function to *kayak.py*:

```
def flightsearchresults(sid,searchid):

  # Removes leading $, commas and converts number to a float
  def parseprice(p):
    return float(p[1:].replace(',',''))

  # Polling loop
  while 1:
    time.sleep(2)

    # Construct URL for polling
    url='http://www.kayak.com/s/basic/flight?'
    url+='searchid=%s&c=5&apimode=1&_sid_=%s&version=1' % (searchid,sid)
    doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))

    # Look for morepending tag, and wait until it is no longer true
    morepending=doc.getElementsByTagName('morepending')[0].firstChild
    if morepending==None or morepending.data=='false': break

  # Now download the complete list
  url='http://www.kayak.com/s/basic/flight?'
  url+='searchid=%s&c=999&apimode=1&_sid_=%s&version=1' % (searchid,sid)
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))

  # Get the various elements as lists
  prices=doc.getElementsByTagName('price')
  departures=doc.getElementsByTagName('depart')
  arrivals=doc.getElementsByTagName('arrive')

  # Zip them together
  return zip([p.firstChild.data.split(' ')[1] for p in departures],
             [p.firstChild.data.split(' ')[1] for p in arrivals],
             [parseprice(p.firstChild.data) for p in prices])
```

Notice that at the end the function just gets all the `price`, `depart`, and `arrive` tags. There will be an equal number of them—one for each flight—so the `zip` function can be used to join them all together into tuples in a big list. The departure and arrival information is given as date and time separated by a space, so the function splits the string to get only the time. The function also converts the price to a float by passing it to `parseprice`.

You can try a real flight search in your Python session to make sure everything is working (remember to change the date to some time in the future):

```
>>> import kayak
>>> sid=kayak.getkayaksession( )
>>> searchid=kayak.flightsearch(sid,'BOS','LGA','11/17/2006')
>>> f=kayak.flightsearchresults(sid,searchid)
>>> f[0:3]
[(u'07:00', u'08:25', 60.3),
 (u'08:30', u'09:49', 60.3),
 (u'06:35', u'07:54', 65.0)]
```

Flights are conveniently returned in order of price, and for flights that are the same price, in order of time. This works out well since, like before, it means that similar solutions are close together. The only requirement to integrate this with the rest of the code is to create a full schedule for all the different people in the Glass family with the same structure that was originally loaded in from the file. This is just a matter of looping over the people in the list and performing the flight search for their outbound and return flights. Add the createschedule function to *kayak.py*:

```
def createschedule(people,dest,dep,ret):
  # Get a session id for these searches
  sid=getkayaksession( )
  flights={}

  for p in people:
    name,origin=p
    # Outbound flight
    searchid=flightsearch(sid,origin,dest,dep)
    flights[(origin,dest)]=flightsearchresults(sid,searchid)

    # Return flight
    searchid=flightsearch(sid,dest,origin,ret)
    flights[(dest,origin)]=flightsearchresults(sid,searchid)

  return flights
```

Now you can try to optimize the flights for the family using actual flight data. The Kayak searches can take a while, so limit the search to just the first two family members to start with. Enter this in your Python session:

```
>>> reload(kayak)
>>> f=kayak.createschedule(optimization.people[0:2],'LGA',
... '11/17/2006','11/19/2006')
>>> optimization.flights=f
>>> domain=[(0,30)]*len(f)
>>> optimization.geneticoptimize(domain,optimization.schedulecost)
770.0
703.0
...
>>> optimization.printschedule(s)
   Seymour       BOS 16:00-17:20 $85.0 19:00-20:28 $65.0
    Franny       DAL 08:00-17:25 $205.0 18:55-00:15 $133.0
```

Congratulations! You've just run an optimization on real live flight data. The search space is much bigger, so it's a good idea to experiment with the maximum velocity and learning rate.

There are many ways this can be expanded. You might combine it with a weather search to optimize for combinations of prices and warm temperatures at potential destinations, or with a hotel search to find destinations with a reasonable combination of flight and hotel prices. There are thousands of sites on the Internet that provide travel destination data that can be used as part of an optimization.

The Kayak API has a limit on searches per day, but it does return links to purchase any flight or hotel directly, which means you can easily incorporate the API into any application.

Optimizing for Preferences

You've seen one example of a problem that optimization can be used to solve, but there are many seemingly unrelated problems that can be attacked using the same methods. Remember, the primary requirements for solving with optimization are that the problem has a defined cost function and that similar solutions tend to yield similar results. Not every problem with these properties will be solvable with optimization, but there's a good chance that optimization will return some interesting results that you hadn't considered.

This section will consider a different problem, one that clearly lends itself to optimization. The general problem is how to allocate limited resources to people who have expressed preferences and make them all as happy as possible (or, depending on their dispositions, annoy them as little as possible).

Student Dorm Optimization

The example problem in this section is that of assigning students to dorms depending on their first and second choices. Although this is a very specific example, it's easy to generalize this case to other problems—the exact same code can be used to assign tables to players in an online card game, assign bugs to developers in a large coding project, or even to assign housework to household members. Once again, the purpose is to take information from individuals and combine it to produce the optimal result.

There are five dorms in our example, each with two spaces available and ten students vying for spots. Each student has first and second choices. Create a new file called *dorm.py* and add the list of dorms and the list of people, along with their top two choices:

```
import random
import math

# The dorms, each of which has two available spaces
dorms=['Zeus','Athena','Hercules','Bacchus','Pluto']

# People, along with their first and second choices
prefs=[('Toby', ('Bacchus', 'Hercules')),
       ('Steve', ('Zeus', 'Pluto')),
       ('Andrea', ('Athena', 'Zeus')),
       ('Sarah', ('Zeus', 'Pluto')),
       ('Dave', ('Athena', 'Bacchus')),
       ('Jeff', ('Hercules', 'Pluto')),
       ('Fred', ('Pluto', 'Athena')),
       ('Suzie', ('Bacchus', 'Hercules')),
       ('Laura', ('Bacchus', 'Hercules')),
       ('Neil', ('Hercules', 'Athena'))]
```

You can see immediately that every person can't have his top choice, since there are only two spots in Bacchus and three people want them. Putting any of these people in their second choice would mean there wouldn't be enough space in Hercules for the people who chose it.

This problem is deliberately small so it's easy to follow, but in real life, this problem might include hundreds or thousands of students competing for many more spots in a larger selection of dorms. Since this example only has about 100,000 possible solutions, it's possible to try them all and see which one is the best. But the number quickly grows to trillions of possibilities when there are four slots in each dorm.

The representation for solutions is a bit trickier for this problem than for the flight problem. You could, in theory, create a list of numbers, one for each student, where each number represents the dorm in which you've put the student. The problem is that this representation doesn't constrain the solution to only two students in each dorm. A list of all zeros would indicate that everyone had been placed in Zeus, which isn't a real solution at all.

One way to resolve this is to make the cost function return a very high value for invalid solutions, but this makes it very difficult for the optimization algorithm to find better solutions because it has no way to determine if it's close to other good or even valid solutions. In general, it's better not to waste processor cycles searching among invalid solutions.

A better way to approach the issue is to find a way to represent solutions so that every one is valid. A valid solution is not necessarily a good solution; it just means that there are exactly two students assigned to each dorm. One way to do this is to think of every dorm as having two slots, so that in the example there are ten slots in total. Each student, in order, is assigned to one of the open slots—the first person can be placed in any one of the ten, the second person can be placed in any of the nine remaining slots, and so on.

The domain for searching has to capture this restriction. Add this line to *dorm.py*:

```
# [(0,9),(0,8),(0,7),(0,6),...,(0,0)]
domain=[(0,(len(dorms)*2)-i-1) for i in range(0,len(dorms)*2)]
```

The code to print the solution illustrates how the slots work. This function first creates a list of slots, two for each dorm. It then loops over every number in the solution and finds the dorm number at that location in the slots list, which is the dorm that a student is assigned to. It prints the student and the dorm, and then it removes that slot from the list so no other student will be given that slot. After the final iteration, the slots list is empty and every student and dorm assignment has been printed. Add this function to *dorm.py*:

```
def printsolution(vec):
  slots=[]
  # Create two slots for each dorm
  for i in range(len(dorms): slots+=[i,i]

  # Loop over each students assignment
  for i in range(len(vec)):
    x=int(vec[i])

    # Choose the slot from the remaining ones
    dorm=dorms[slots[x]]
    # Show the student and assigned dorm
    print prefs[i][0],dorm
    # Remove this slot
    del slots[x]
```

In your Python session, you can import this and try printing a solution:

```
>>> import dorm
>>> dorm.printsolution([0,0,0,0,0,0,0,0,0,0])
Toby Zeus
Steve Zeus
Andrea Athena
Sarah Athena
Dave Hercules
Jeff Hercules
Fred Bacchus
Suzie Bacchus
Laura Pluto
Neil Pluto
```

If you change the numbers around to view different solutions, remember that each number must stay in the appropriate range. The first item in the list can be between 0 and 9, the second between 0 and 8, etc. If you set one of the numbers outside the appropriate range, the function will throw an exception. Since the optimization functions will keep the numbers in the ranges specified in the domain parameter, this won’t be a problem when optimizing.

The Cost Function

The cost function works in a way that is similar to the print function. A list of slots is constructed and slots are removed as they are used up. The cost is calculated by comparing a student's current dorm assignment to his top two choices. The total cost will increase by 0 if the student is currently assigned to his top choice, by 1 if he is assigned to his second choice, and by 3 if he is not assigned to either of his choices:

```
def dormcost(vec):
  cost=0
  # Create list a of slots
  slots=[0,0,1,1,2,2,3,3,4,4]

  # Loop over each student
  for i in range(len(vec)):
    x=int(vec[i])
    dorm=dorms[slots[x]]
    pref=prefs[i][1]
    # First choice costs 0, second choice costs 1
    if pref[0]==dorm: cost+=0
    elif pref[1]==dorm: cost+=1
    else: cost+=3
    # Not on the list costs 3

    # Remove selected slot
    del slots[x]

  return cost
```

A useful rule when creating a cost function is, if possible, to make the perfect solution (which in this example is everyone being assigned to their top choice) have a cost of zero. In this case, you've already determined that the perfect solution is impossible, but knowing that its cost is zero gives you an idea of how close you are to it. The other advantage of this rule is that you can tell an optimization algorithm to stop searching for better solutions if it ever finds a perfect solution.

Running the Optimization

With a solution representation, a cost function, and a function to print the results, you have enough to run the optimization functions that you defined earlier. Enter the following in your Python session:

```
>>> reload(dorm)
>>> s=optimization.randomoptimize(dorm.domain,dorm.dormcost)
>>> dorm.dormcost(s)
18
>>> optimization.geneticoptimize(dorm.domain,dorm.dormcost)
13
10
...
4
>>> dorm.printsolution(s)
```

```
    Toby Athena
    Steve Pluto
    Andrea Zeus
    Sarah Pluto
    Dave Hercules
    Jeff Hercules
    Fred Bacchus
    Suzie Bacchus
    Laura Athena
    Neil Zeus
```

Again, you can tweak the parameters to see if you can make the genetic optimization find a good solution more quickly.

Network Visualization

The final example in this chapter shows another way in which optimization can be used on problems that are completely unrelated to one another. In this case, the problem is the visualization of networks. A network in this case is any set of things that are connected together. A good example in online applications is a social network like MySpace, Facebook, or LinkedIn, where people are connected because they are friends or have a professional relationship. Each member of the site chooses to whom they are connected, and collectively this creates a network of people. It is interesting to visualize such networks to determine their structure, perhaps in order to find the people who are connectors (those who know a lot of people or who serve as a link between otherwise self-contained cliques).

The Layout Problem

When drawing a network to visualize a big group of people and the links between them, one problem is deciding where each name (or icon) should be placed in the picture. For example, consider the network in Figure 5-7.

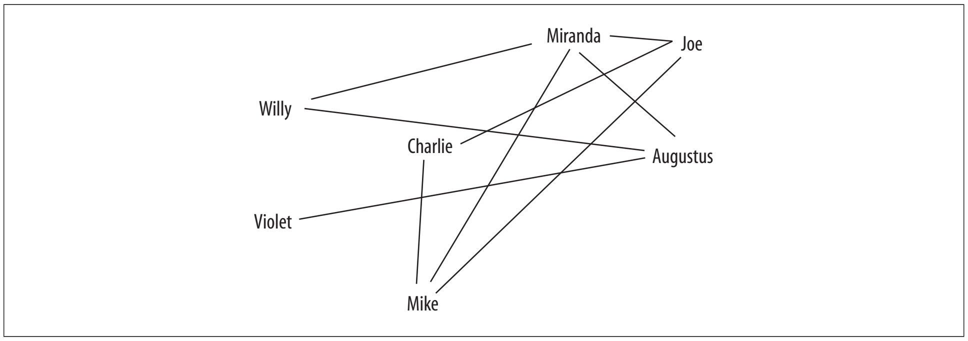


Figure 5-7. A confusing network layout

In this figure, you can see that Augustus is friends with Willy, Violet, and Miranda. But the layout of the network is a bit messy, and adding more people would make it very confusing. A much cleaner layout is shown in Figure 5-8.

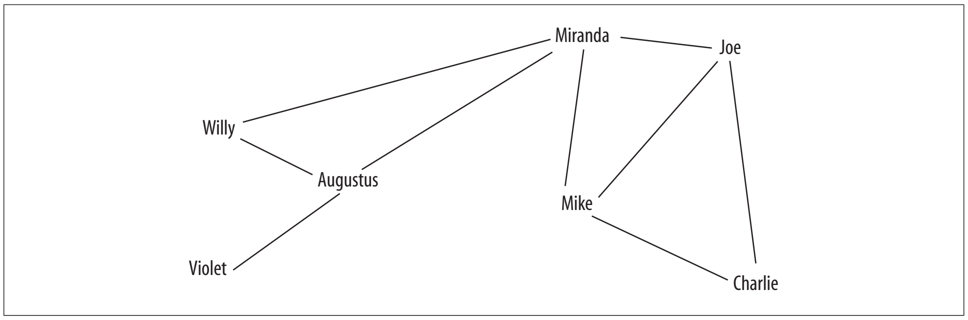


Figure 5-8. A clean network layout

This section will look at how optimization can be used to create better, less confusing visuals. To begin, create a new file called *socialnetwork.py* and add some facts about a subsection of the social network:

```
import math

people=['Charlie','Augustus','Veruca','Violet','Mike','Joe','Willy','Miranda']

links=[('Augustus', 'Willy'),
       ('Mike', 'Joe'),
       ('Miranda', 'Mike'),
       ('Violet', 'Augustus'),
       ('Miranda', 'Willy'),
       ('Charlie', 'Mike'),
       ('Veruca', 'Joe'),
       ('Miranda', 'Augustus'),
       ('Willy', 'Augustus'),
       ('Joe', 'Charlie'),
       ('Veruca', 'Augustus'),
       ('Miranda', 'Joe')]
```

The goal here is to create a program that can take a list of facts about who is friends with whom and generate an easy-to-interpret network diagram. This is usually done with a *mass-and-spring* algorithm. This type of algorithm is modeled on physics because the different nodes exert a push on each other and try to move apart, while the links try to pull connected nodes closer together. Thus, the network slowly assumes a layout where unconnected nodes are pushed apart and connected nodes are pulled close together—but not too close together.

Unfortunately, the mass-and-spring algorithm doesn't stop lines from crossing. In a network with a great number of links, this makes it difficult to see which nodes are connected because visually tracking the lines as they cross can be tricky. However, when you use optimization to create the layout, all you need to do is decide on a cost function and then try to minimize it. In this case, one interesting cost function to try is the number of lines that cross each other.

Counting Crossed Lines

In order to use the same optimizing functions that were defined earlier, it's necessary to represent a solution as a list of numbers. Fortunately, this particular problem is represented as a list of numbers very easily—every node has an x and y coordinate, so the coordinates for all the nodes can be put into a long list:

```
sol=[120,200,250,125 ...
```

In this solution, Charlie is placed at (120,200), Augustus at (250,125), and so on.

Right now, the new cost function will simply count the number of lines that cross each other. The derivation of the formula for two lines crossing is a bit beyond the scope of this chapter, but the basic idea is to calculate the fraction of the line where each line is crossed. If this fraction is between 0 (one end of the line) and 1 (the other end), for both lines, then they cross each other. If the fraction is not between 0 and 1, then the lines do not cross.

This function loops through every pair of links and uses the current coordinates of their endpoints to determine whether they cross. If they do, the function adds 1 to the total score. Add `crosscount` to *socialnetwork.py*:

```
def crosscount(v):
  # Convert the number list into a dictionary of person:(x,y)
  loc=dict([(people[i],(v[i*2],v[i*2+1])) for i in range(0,len(people))])
  total=0

  # Loop through every pair of links
  for i in range(len(links)):
    for j in range(i+1,len(links)):

      # Get the locations
      (x1,y1),(x2,y2)=loc[links[i][0]],loc[links[i][1]]
      (x3,y3),(x4,y4)=loc[links[j][0]],loc[links[j][1]]

      den=(y4-y3)*(x2-x1)-(x4-x3)*(y2-y1)

      # den==0 if the lines are parallel
      if den==0: continue

      # Otherwise ua and ub are the fraction of the
      # line where they cross
      ua=((x4-x3)*(y1-y3)-(y4-y3)*(x1-x3))/den
      ub=((x2-x1)*(y1-y3)-(y2-y1)*(x1-x3))/den
```

```
        # If the fraction is between 0 and 1 for both lines
        # then they cross each other
        if ua>0 and ua<1 and ub>0 and ub<1:
          total+=1
  return total
```

The domain for this search is the range for each coordinate. For this example, you can assume that the network will be laid out in a 400×400 image, so the domain will be a little less than that to allow for a slight margin. Add this line to the end of *socialnetwork.py*:

```
domain=[(10,370)]*(len(people)*2)
```

Now you can try actually running some of the optimizations to find a solution where very few lines cross. Import *socialnetwork.py* to your Python session and try a couple of the optimization algorithms:

```
>>> import socialnetwork
>>> import optimization
>>> sol=optimization.randomoptimize(socialnetwork.domain,socialnetwork.crosscount)
>>> socialnetwork.crosscount(sol)
12
>>> sol=optimization.annealingoptimize(socialnetwork.domain,
    socialnetwork.crosscount,step=50,cool=0.99)
>>> socialnetwork.crosscount(sol)
1
>>> sol
[324, 190, 241, 329, 298, 237, 117, 181, 88, 106, 56, 10, 296, 370, 11, 312]
```

Simulated annealing is likely to find a solution where very few of the lines cross, but the list of coordinates is difficult to interpret. The next section will show you how to automatically draw the network.

Drawing the Network

You'll need the Python Imaging Library that was used in Chapter 3. If you haven't installed it yet, please consult Appendix A for instructions on getting the latest version and installing it with your Python instance.

The code for drawing the network is quite straightforward. All the code has to do is create an image, draw the links between the different people, and then draw the nodes for the people. The people's names are drawn afterward so that the lines don't cover them. Add this function to *socialnetwork.py*:

```
def drawnetwork(sol):
  # Create the image
  img=Image.new('RGB',(400,400),(255,255,255))
  draw=ImageDraw.Draw(img)

  # Create the position dict
  pos=dict([(people[i],(sol[i*2],sol[i*2+1])) for i in range(0,len(people))])
```

```
    # Draw Links
    for (a,b) in links:
      draw.line((pos[a],pos[b]),fill=(255,0,0))

    # Draw people
    for n,p in pos.items( ):
      draw.text(p,n,(0,0,0))

    img.show( )
```

To run this function in your Python session, just reload the module and call this function on your solution:

```
>>> reload(socialnetwork)
>>> drawnetwork(sol)
```

Figure 5-9 shows one possible outcome of the optimization.

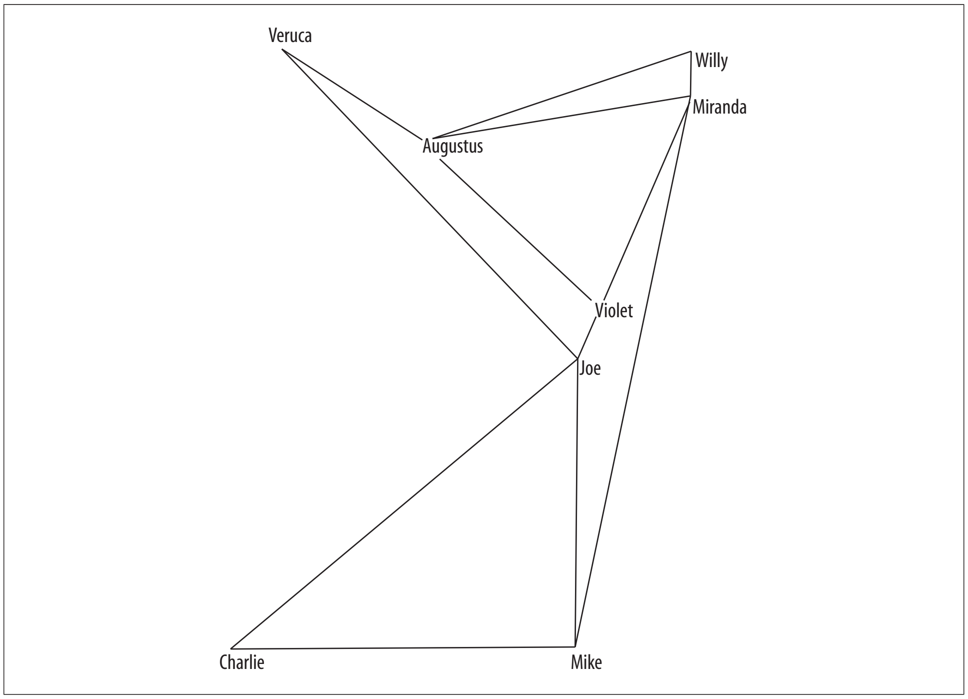


Figure 5-9. Layout resulting from a no-crossed-lines optimization

Of course, your solution will look different from this. Sometimes the solution will look pretty wacky—since the objective is just to minimize the number of crossed lines, the cost function never penalizes the layout for things like very tight angles between the lines or two nodes being very close together. In this respect, optimization is like a genie who grants your wishes very literally, so it's always important to be clear about what you want. There is often a solution that fits the original criteria of "best" but looks nothing like what you had in mind.

A simple way to penalize a solution that has put two nodes too close together is to calculate the distance between the nodes and divide by a desired minimum distance. You can add this code to the end of crosscount (before the return statement) to provide an additional penalty.

```
for i in range(len(people)):
  for j in range(i+1,len(people)):
    # Get the locations of the two nodes
    (x1,y1),(x2,y2)=loc[people[i]],loc[people[j]]

    # Find the distance between them
    dist=math.sqrt(math.pow(x1-x2,2)+math.pow(y1-y2,2))
    # Penalize any nodes closer than 50 pixels
    if dist<50:
      total+=(1.0-(dist/50.0))
```

This creates a higher cost for every pair of nodes that is less than 50 pixels apart, in proportion to how close together they are. If they are in exactly the same place, the penalty is 1. Run the optimization again to see if this results in a more spread-out layout.

Other Possibilities

This chapter has shown three completely different applications for optimization algorithms, but that's only a small fraction of what is possible. As stated throughout the chapter, the important steps are deciding on a representation and a cost function. If you can do these things, there's a good chance you can use optimization to find solutions to your problem.

An interesting activity might be to take a large group of people and divide them into teams in which the skills of the members are evenly divided. In a trivia contest, it might be desirable to create teams from a set of people so that each team has adequate knowledge of sports, history, literature, and television. Another possibility is to assign tasks in group projects by taking a combination of people's skills into account. Optimization can determine the best way to divide the tasks so that the task list is completed in the shortest possible time.

Given a long list of web sites tagged with keywords, it might be interesting to find an optimal group of web sites for a user-supplied set of keywords. The optimal group would contain a set of web sites that don't have many keywords in common with each other but represent as many of the user-supplied keywords as possible.

Exercises

1. *Group travel cost function*. Add total flight time as a cost equal to $0.50 per minute on the plane. Next try adding a penalty of $20 for making anyone get to the airport before 8 a.m.
2. *Annealing starting points*. The outcome of simulated annealing depends heavily on the starting point. Build a new optimization function that does simulated annealing from multiple starting solutions and returns the best one.
3. *Genetic optimization stopping criteria*. A function in this chapter runs the genetic optimizer for a fixed number of iterations. Change it so that it stops when there has been no improvement in any of the best solutions for 10 iterations.
4. *Round-trip pricing*. The function for getting flight data from Kayak right now only looks for one-way flights. Prices are probably cheaper when buying round-trip tickets. Modify the code to get round-trip prices, and modify the cost function to use a price lookup for a particular pair of flights instead of just summing their one-way prices.
5. *Pairing students*. Imagine if instead of listing dorm preferences, students had to express their preferences for a roommate. How would you represent solutions to pairing students? What would the cost function look like?
6. *Line angle penalization*. Add an additional cost to the network layout algorithm cost function when the angle between two lines attached to the same person is very small. (Hint: you can use the *vector cross-product*.)

CHAPTER 11

Evolving Intelligence

Throughout this book you've seen a number of different problems, and in each case you used an algorithm that was suited to solve that particular problem. In some of the examples, you had to tweak the parameters or use optimization to search for a good set of parameters. This chapter will look at a different way to approach problems. Instead of choosing an algorithm to apply to a problem, you'll make a program that attempts to automatically build the best program to solve a problem. Essentially, you'll be creating an algorithm that creates algorithms.

To do this, you will use a technique called *genetic programming*. Since this is the last chapter in which you'll learn a completely new type of algorithm, I've picked a topic that is new, exciting, and being actively researched. This chapter is a little different from the others because it doesn't use any open APIs or public datasets, and because programs that can modify themselves based on their interactions with many people are an interesting and different kind of collective intelligence. Genetic programming is a very large topic about which many books have been written, so you'll only get an introduction here, but I hope it's enough to get you excited about the possibilities and perhaps to research and experiment on your own.

The two problems in this chapter are recreating a mathematical function given a dataset, and automatically creating an AI (artificial intelligence) player for a simple board game. This is just a very small sampling of the possibilities of genetic programming—computational power is really the only constraint on the types of problems it can be used to solve.

What Is Genetic Programming?

Genetic programming is a machine-learning technique inspired by the theory of biological evolution. It generally works by starting with a large set of programs (referred to as the *population*), which are either randomly generated or hand-designed and are known to be somewhat good solutions. The programs are then made to compete in some user-defined task. This may be a game in which the programs compete against each other directly, or it may be an individual test to see which program performs better. After the competition, a ranked list of the programs from best to worst can be determined.

Next—and here's where evolution comes in—the best programs are replicated and modified in two different ways. The simpler way is *mutation*, in which certain parts of the program are altered very slightly in a random manner in the hope that this will make a good solution even better. The other way to modify a program is through *crossover* (sometimes referred to as *breeding*), which involves taking a portion of one of the best programs and replacing it with a portion of one of the other best programs. This replication and modification procedure creates many new programs that are based on, but different from, the best programs.

At each stage, the quality of the programs is calculated using a *fitness function*. Since the size of the population is kept constant, many of the *worst* programs are eliminated from the population to make room for the new programs. The new population is referred to as "the next generation," and the whole procedure is then repeated. Because the best programs are being kept and modified, it is expected that with each generation they will get better and better, in much the same way that teenagers can be smarter than their parents.

New generations are created until a termination condition is reached, which, depending on the problem, can be that:

- The perfect solution has been found.
- A good enough solution has been found.
- The solution has not improved for several generations.
- The number of generations has reached a specified limit.

For some problems, such as determining a mathematical function that correctly maps a set of inputs to an output, a perfect solution is possible. For others, such as a board game, there may not be a perfect solution, since the quality of a solution depends on the strategy of the program's adversary.

An overview of the genetic programming process is shown as a flowchart in Figure 11-1.

Genetic Programming Versus Genetic Algorithms

Chapter 5 introduced a related set of algorithms known as *genetic algorithms*. Genetic algorithms are an optimization technique that use the idea of evolutionary pressure to choose the best result. With any form of optimization, you have already selected an algorithm or metric and you're simply trying to find the best parameters for it.

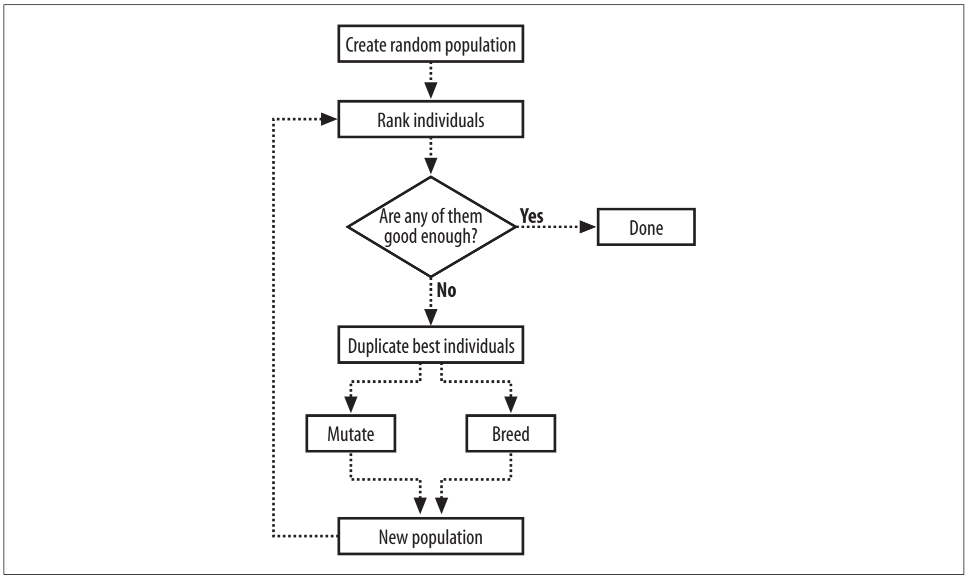


Figure 11-1. Genetic programming overview

Successes of Genetic Programming

Genetic programming has been around since the 1980s, but it is very computationally intensive, and with the computing power that was available at the time, it couldn't be used for anything more than simple problems. As computers have gotten faster, however, people have been able to apply genetic programming to sophisticated problems. Many previously patented inventions have been rediscovered or improved using genetic programming, and recently several new patentable inventions have been designed by computers.

The genetic programming technique has been applied in designing antennas for NASA, and in photonic crystals, optics, quantum computing systems, and other scientific inventions. It has also been used to develop programs for playing many games, such as chess and backgammon. In 1998, researchers from Carnegie Mellon University entered a robot team that was programmed entirely using genetic programming into the RoboCup soccer contest, and placed in the middle of the pack.

Like optimization, genetic programming requires a way to measure how good a solution is; but unlike optimization, the solutions are not just a set of parameters being applied to a given algorithm. Instead, the algorithm itself and all its parameters are designed automatically by means of evolutionary pressure.

Programs As Trees

In order to create programs that can be tested, mutated, and bred, you'll need a way to represent and run them from within your Python code. The representation has to lend itself to easy modification and, more importantly, has to be guaranteed to be an actual program—which means generating random strings and trying to treat them as Python code won't work. Researchers have come up with a few different ways to represent programs for genetic programming, and the most commonly used is a tree representation.

Most programming languages, when compiled or interpreted, are first turned into a *parse tree*, which is very similar to what you'll be working with here. (The programming language Lisp and its variants are essentially ways of entering a parse tree directly.) An example of a parse tree is shown in Figure 11-2.

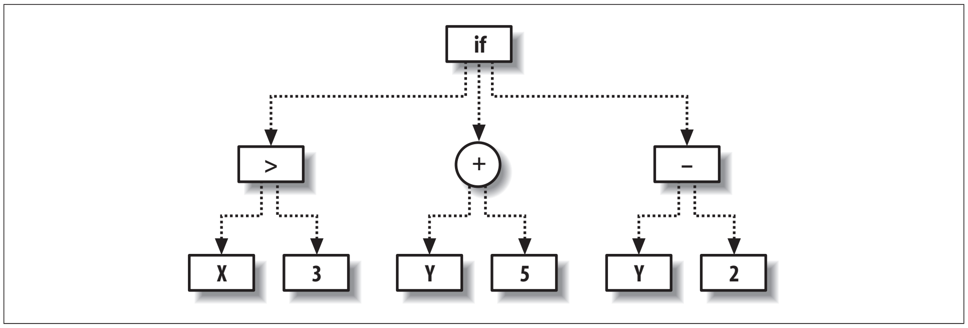


Figure 11-2. Example program tree

Each node represents either an operation on its child nodes or an endpoint, such as a parameter with a constant value. For example, the circular node is a sum operation on its two branches, in this case, the values Y and 5. Once this point is evaluated, it is given to the node above it, which in turn applies its own operation to its branches. You'll also notice that one of the nodes has the operation "if," which specifies that if its leftmost branch evaluates to true, return its center branch; if it doesn't, return its rightmost branch.

Traversing the complete tree, you can see that it corresponds to the Python function:

```
def func(x,y)
  if x>3:
    return y + 5
  else:
    return y - 2
```

At first, it might appear that these trees can only be used to build very simple functions. There are two things to consider here—first, the nodes that compose the tree can potentially be very complex functions, such as distance measures or

Gaussians. The second thing is that trees can be made recursive by referring to nodes higher up in the tree. Creating trees like this allows for loops and other more complicated control structures.

Representing Trees in Python

You're now ready to construct tree programs in Python. The trees are made up of nodes, which, depending on the functions associated with them, have some number of child nodes. Some of the nodes will return parameters passed to the program, others will return constants, and the most interesting ones will return operations on their child nodes.

Create a new file called *gp.py* and create four new classes called `fwrapper`, `node`, `paramnode`, and `constnode`:

```
from random import random,randint,choice
from copy import deepcopy
from math import log

class fwrapper:
  def __init__(self,function,childcount,name):
    self.function=function
    self.childcount=childcount
    self.name=name

class node:
  def __init__(self,fw,children):
    self.function=fw.function
    self.name=fw.name
    self.children=children

  def evaluate(self,inp):
    results=[n.evaluate(inp) for n in self.children]
    return self.function(results)

class paramnode:
  def __init__(self,idx):
    self.idx=idx

  def evaluate(self,inp):
    return inp[self.idx]

class constnode:
  def __init__(self,v):
    self.v=v
  def evaluate(self,inp):
    return self.v
```

The classes here are:

`fwrapper`
: A wrapper for the functions that will be used on function nodes. Its member variables are the name of the function, the function itself, and the number of parameters it takes.

`node`
: The class for function nodes (nodes with children). This is initialized with an `fwrapper`. When `evaluate` is called, it evaluates the child nodes and then applies the function to their results.

`paramnode`
: The class for nodes that only return one of the parameters passed to the program. Its `evaluate` method returns the parameter specified by `idx`.

`constnode`
: Nodes that return a constant value. The `evaluate` method simply returns the value with which it was initialized.

You'll also want some functions for the nodes to apply. To do this, you have to create functions and then give them names and parameter counts using `fwrapper`. Add this list of functions to *gp.py*:

```
addw=fwrapper(lambda l:l[0]+l[1],2,'add')
subw=fwrapper(lambda l:l[0]-l[1],2,'subtract')
mulw=fwrapper(lambda l:l[0]*l[1],2,'multiply')

def iffunc(l):
  if l[0]>0: return l[1]
  else: return l[2]
ifw=fwrapper(iffunc,3,'if')

def isgreater(l):
  if l[0]>l[1]: return 1
  else: return 0
gtw=fwrapper(isgreater,2,'isgreater')

flist=[addw,mulw,ifw,gtw,subw]
```

Some of the simpler functions such as `add` and `subtract` can be defined inline using lambda, while others require you to define the function in a separate block. In each case, they have been wrapped in an `fwrapper` with their names and the number of parameters required. The last line creates a list of all the functions so that later they can easily be chosen at random.

Building and Evaluating Trees

You can now construct the program tree shown in Figure 11-2 using the `node` class you just created. Add the `exampletree` function to *gp.py* to create the tree:

```
def exampletree( ):
  return node(ifw,[
                  node(gtw,[paramnode(0),constnode(3)]),
                  node(addw,[paramnode(1),constnode(5)]),
                  node(subw,[paramnode(1),constnode(2)]),
                  ]
              )
```

Start up a Python session to test your program:

```
>>> import gp
>>> exampletree=gp.exampletree( )
>>> exampletree.evaluate([2,3])
1
>>> exampletree.evaluate([5,3])
8
```

The program successfully performs the same function as the equivalent code block, so you've managed to build a mini tree-based language and interpreter within Python. This language can be easily extended with more node types, and it will serve as the basis for understanding genetic programming in this chapter. Try building a few other simple program trees to make sure you understand how they work.

Displaying the Program

Because you'll be creating program trees automatically and won't know what their structure looks like, it's important to have a way to display them so that you can easily interpret them. Fortunately the design of the node class means every node has a string representing the name of its function, so a display function simply has to return that string and the display strings of the child nodes. To make it easier to read, the display should also indent the child nodes so you can visually identify the parent-child relationships in the tree.

Create a new method in the node class called display, which shows a string representation of the tree:

```
  def display(self,indent=0):
    print (' '*indent)+self.name
    for c in self.children:
      c.display(indent+1)
```

You'll also need to create a display method for the paramnode class, which simply prints the index of the parameter it returns:

```
  def display(self,indent=0):
    print '%sp%d' % (' '*indent,self.idx)
```

And finally, one for the constnode class:

```
  def display(self,indent=0):
    print '%s%d' % (' '*indent,self.v)
```

Use these methods to print out the tree:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> exampletree=gp.exampletree( )
>>> exampletree.display( )
if
 isgreater
  p0
  3
 add
  p1
  5
 subtract
  p1
  2
```

If you've read Chapter 7, you'll notice that this is similar to the way in which decision trees were displayed in that chapter. Chapter 7 also shows how to display those trees graphically for a cleaner, easier-to-read output. If you feel so inclined, you can use the same idea to build a graphical display of your tree programs.

Creating the Initial Population

Although it's possible to hand-create programs for genetic programming, most of the time the initial population consists of a set of random programs. This makes the process easier to start, since it's not necessary to design several programs that almost solve a problem. It also creates much more *diversity* in the initial population—a set of programs designed by a single programmer to solve a problem are likely to be very similar, and although they may give answers that are almost correct, the ideal solution make look quite different. You'll learn more about the importance of diversity shortly.

Creating a random program consists of creating a root node with a random associated function and then creating as many random child nodes as necessary, which in turn may have their own associated random child nodes. Like most functions that work with trees, this is most easily defined recursively. Add a new function, `makerandomtree`, to *gp.py*:

```
def makerandomtree(pc,maxdepth=4,fpr=0.5,ppr=0.6):
  if random( )<fpr and maxdepth>0:
    f=choice(flist)
    children=[makerandomtree(pc,maxdepth-1,fpr,ppr)
              for i in range(f.childcount)]
    return node(f,children)
  elif random( )<ppr:
    return paramnode(randint(0,pc-1))
  else:
    return constnode(randint(0,10))
```

This function creates a node with a random function and then looks to see how many child nodes this function requires. For every child node required, the function calls itself to create a new node. In this way an entire tree is constructed, with branches ending only if the function requires no more child nodes (that is, if the function returns a constant or an input variable). The parameter pc, used throughout this chapter, is the number of parameters that the tree will take as input. The parameter fpr gives the probability that the new node created will be a function node, and ppr gives that probability that it will be a paramnode if it is not a function node.

Try out this function in your Python session to build a few programs, and see what sort of results you get with different variables:

```
>>> random1=gp.makerandomtree(2)
>>> random1.evaluate([7,1])
7
>>> random1.evaluate([2,4])
2
>>> random2=gp.makerandomtree(2)
>>> random2.evaluate([5,3])
1
>>> random2.evaluate([5,20])
0
```

If all of a program's terminating nodes are constants, the program will not actually reference the input parameters at all, so the result will be the same no matter what input you pass to it. You can use the function defined in the previous section to display the randomly generated trees:

```
>>> random1.display( )
p0
>>> random2.display( )
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   2
```

You'll see that some of the trees get quite deep, since each branch will keep growing until it hits a zero-child node. This is why it's important that you include a maximum depth constraint; otherwise, the trees can get very large and potentially overflow the stack.

Testing a Solution

You would now have everything you'd need to build programs automatically, if you could just generate random programs until one is correct. Obviously, this would be ridiculously impractical because there are infinite possible programs and it's highly unlikely that you would stumble across a correct one in any reasonable time frame. However, at this point it is worth looking at ways to test a solution to see if it's correct, and if it's not, to determine how close it is.

A Simple Mathematical Test

One of the easiest tests for genetic programming is to reconstruct a simple mathematical function. Imagine you were given a table of inputs and an output that looked like Table 11-1.

Table 11-1. Data and result for an unknown function

X	Y	Result
26	35	829
8	24	141
20	1	467
33	11	1215
37	16	1517

There is some function that maps X and Y to the result, but you're not told what it is. A statistician might see this and try to do a regression analysis, but that requires guessing the structure of the formula first. Another option is to build a predictive model using k-nearest neighbors as you did in Chapter 8, but that requires keeping all the data. In some cases, you just need a formula, perhaps to codify in another much simpler program or to describe to other people what's going on.

I'm sure you're in suspense, so I'll tell you what the function is. Add `hiddenfunction` to *gp.py*:

```
def hiddenfunction(x,y):
    return x**2+2*y+3*x+5
```

You're going to use this function to build a dataset against which you can test your generated programs. Add a new function, `buildhiddenset`, which creates the dataset:

```
def buildhiddenset( ):
  rows=[]
  for i in range(200):
    x=randint(0,40)
    y=randint(0,40)
    rows.append([x,y,hiddenfunction(x,y)])
  return rows
```

And use this to create a dataset in your Python session:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> hiddenset=gp.buildhiddenset( )
```

Of course, you know what the function used to generate the dataset looks like, but the real test is whether genetic programming can reproduce it without being told.

Measuring Success

As with optimization, it's necessary to come up with a way to measure how good a solution is. In this case, you're testing a program against a numerical outcome, so an easy way to test a program is to see how close it gets to the correct answers for the dataset. Add `scorefunction` to *gp.py*:

```
def scorefunction(tree,s):
  dif=0
  for data in s:
    v=tree.evaluate([data[0],data[1]])
    dif+=abs(v-data[2])
  return dif
```

This function checks every row in the dataset, calculating the output from the function and comparing it to the real result. It adds up all the differences, giving lower values for better programs—a return value of 0 indicates that the program got every result correct. You can now test some of your generated programs in your Python session to see how they stack up:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> gp.scorefunction(random2,hiddenset)
137646
>>> gp.scorefunction(random1,hiddenset)
125489
```

Since you only generated a few programs and they were generated completely randomly, the chance that one of them is actually the correct function is vanishingly small. (If one of your programs is the correct function, I suggest that you put the book down and go buy yourself a lottery ticket.) However, you now have a way to test how well a program performs on predicting a mathematical function, which is important for deciding which programs make it to the next generation.

Mutating Programs

After the best programs are chosen, they are replicated and modified for the next generation. As mentioned earlier, mutation takes a single program and alters it slightly. The tree programs can be altered in a number of ways—by changing the function on a node or by altering its branches. A function that changes the number of required child nodes either deletes or adds new branches, as shown in Figure 11-3.

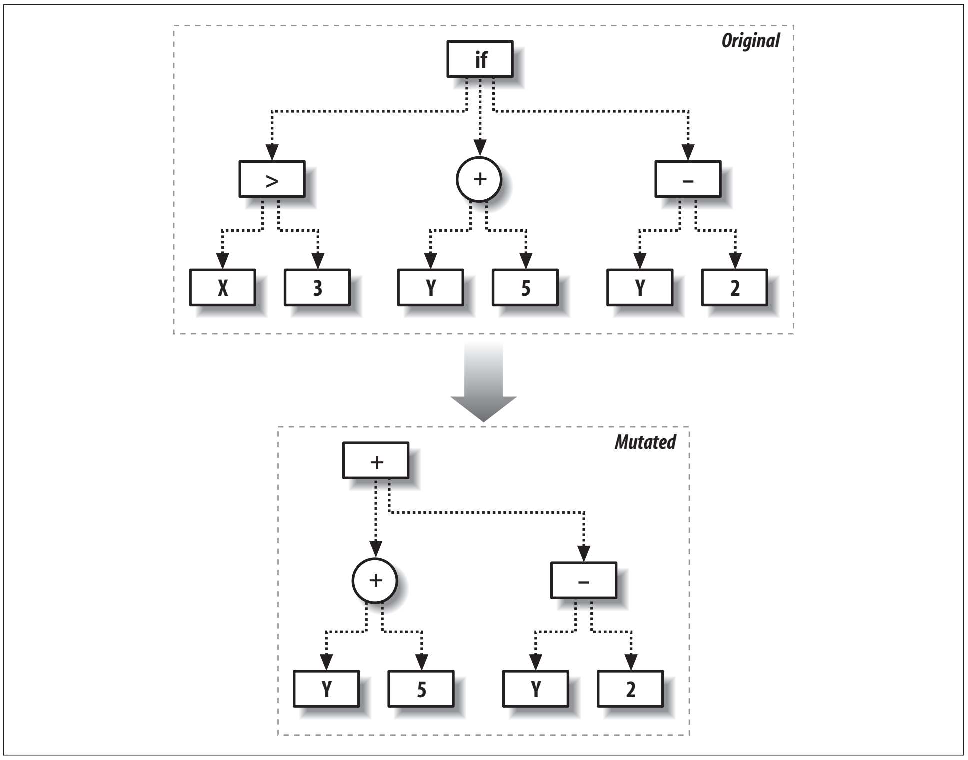


Figure 11-3. Mutation by changing node functions

The other way to mutate is by replacing a subtree with an entirely new one, as shown in Figure 11-4.

Mutation is not something that should be done too much. You would not, for instance, mutate the majority of nodes in a tree. Instead, you can assign a relatively small probability that any node will be modified. Beginning at the top of the tree, if a random number is lower than that probability, the node is mutated in one of the ways described above; otherwise, the test is performed again on its child nodes.

To keep things simple, the code given here only performs the second kind of mutation. Create a new function called `mutate` to perform this operation:

```
def mutate(t,pc,probchange=0.1):
  if random( )<probchange:
    return makerandomtree(pc)
  else:
    result=deepcopy(t)
    if isinstance(t,node):
      result.children=[mutate(c,pc,probchange) for c in t.children]
    return result
```

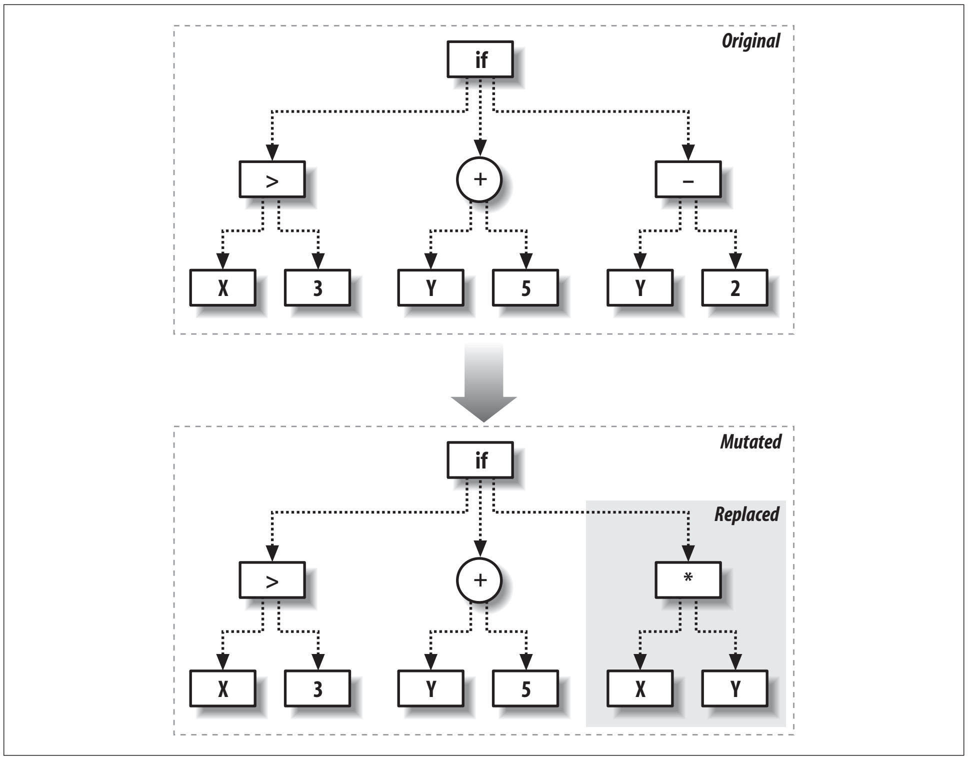


Figure 11-4. Mutation by replacing subtrees

This function begins at the top of the tree and decides whether the node should be altered. If not, it calls `mutate` on the child nodes of the tree. It's possible that the entire tree will be mutated, and it's also possible to traverse the entire tree without changing it.

Try running `mutate` a few times on the randomly generated programs you built earlier, and see how it modifies the trees:

```
>>> random2.display()
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   2
```

```
>>> muttree=gp.mutate(random2,2)
>>> muttree.display( )
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   p1
```

See if the result of scorefunction has changed significantly, for better or worse, after the tree has been mutated:

```
>>> gp.scorefunction(random2,hiddenset)
125489
>>> gp.scorefunction(muttree,hiddenset)
125479
```

Remember that the mutations are random, and they aren't necessarily directed toward improving the solution. The hope is simply that some will improve the result. These changes will be used to continue, and over several generations the best solution will eventually be found.

Crossover

The other type of program modification is crossover or breeding. This involves taking two successful programs and combining them to create a new program, usually by replacing a branch from one with a branch from another. Figure 11-5 shows an example of how this works.

The function for performing a crossover takes two trees as inputs and traverses down both of them. If a randomly selected threshold is reached, the function returns a copy of the first tree with one of its branches replaced by a branch in the second tree. By traversing both trees at once, the crossover happens at approximately the same level on each tree. Add the crossover function to *gp.py*:

```
def crossover(t1,t2,probswap=0.7,top=1):
  if random( )<probswap and not top:
    return deepcopy(t2)
  else:
    result=deepcopy(t1)
    if isinstance(t1,node) and isinstance(t2,node):
      result.children=[crossover(c,choice(t2.children),probswap,0)
                       for c in t1.children]
    return result
```

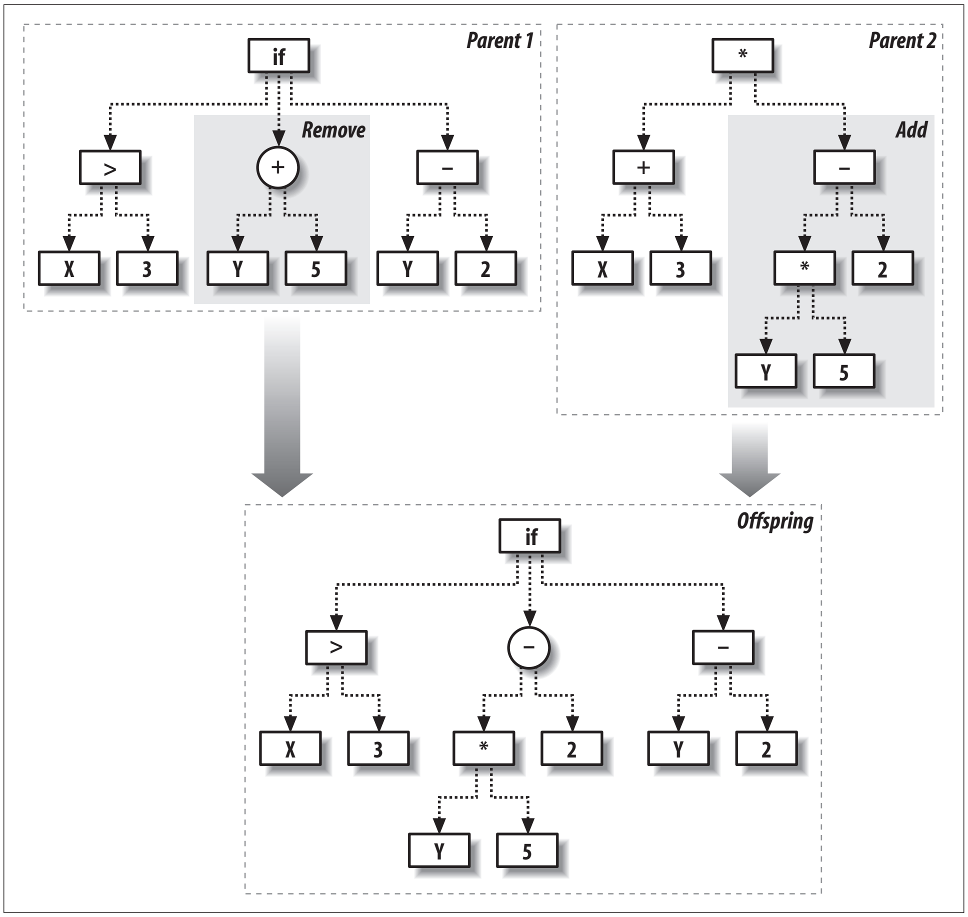


Figure 11-5. Crossover operation

Try crossover on a few of the randomly generated programs. See what they look like after the crossover, and see if crossing over two of the best programs occasionally leads to a better program:

```
>>> random1=gp.makerandomtree(2)
>>> random1.display( )
multiply
 subtract
  p0
  8
 isgreater
  p0
  isgreater
   p1
   5
```

```
>>> random2=gp.makerandomtree(2)
>>> random2.display( )
if
 8
 p1
 2
>>> cross=gp.crossover(random1,random2)
>>> cross.display( )
multiply
 subtract
  p0
  8
 2
```

You'll probably notice that swapping out branches can radically change what the program does. You may also notice that programs may be close to being correct for completely different reasons, so merging them produces a result that's very different from either of its predecessors. Again, the hope is that some crossovers will improve the solution and be kept around for the next generation.

Building the Environment

Armed with a measure of success and two methods of modifying the best programs, you're now ready to set up a competitive environment in which programs can evolve. The steps are shown in the flowchart in Figure 11-1. Essentially, you create a set of random programs and select the best ones for replication and modification, repeating this process until some stopping criteria is reached.

Create a new function called evolve to carry out this procedure:

```
def evolve(pc,popsize,rankfunction,maxgen=500,
           mutationrate=0.1,breedingrate=0.4,pexp=0.7,pnew=0.05):
  # Returns a random number, tending towards lower numbers. The lower pexp
  # is, more lower numbers you will get
  def selectindex( ):
    return int(log(random( ))/log(pexp))

  # Create a random initial population
  population=[makerandomtree(pc) for i in range(popsize)]
  for i in range(maxgen):
    scores=rankfunction(population)
    print scores[0][0]
    if scores[0][0]==0: break

    # The two best always make it
    newpop=[scores[0][1],scores[1][1]]
    # Build the next generation
    while len(newpop)<popsize:
      if random( )>pnew:
        newpop.append(mutate(
                      crossover(scores[selectindex( )][1],
                                 scores[selectindex( )][1],
                                probswap=breedingrate),
                        pc,probchange=mutationrate))
      else:
      # Add a random node to mix things up
        newpop.append(makerandomtree(pc))

    population=newpop
  scores[0][1].display( )
  return scores[0][1]
```

This function creates an initial random population. It then loops up to maxgen times, each time calling rankfunction to rank the programs from best to worst. The best program is automatically passed through to the next generation unaltered, which is sometimes referred to as *elitism*. The rest of the next generation is constructed by randomly choosing programs that are near the top of the ranking, and then breeding and mutating them. This process repeats until either a program has a perfect score of 0 or maxgen is reached.

The function has several parameters, which are used to control various aspects of the environment. They are:

rankfunction
: The function used on the list of programs to rank them from best to worst.

mutationrate
: The probability of a mutation, passed on to mutate.

breedingrate
: The probability of crossover, passed on to crossover.

popsize
: The size of the initial population.

probexp
: The rate of decline in the probability of selecting lower-ranked programs. A higher value makes the selection process more stringent, choosing only programs with the best ranks to replicate.

probnew
: The probability when building the new population that a completely new, random program is introduced. probexp and probnew will be discussed further in the upcoming section "The Importance of Diversity."

The final thing you'll need before beginning the evolution of your programs is a way to rank programs based on the result of scorefunction. In *gp.py*, create a new function called getrankfunction, which returns a ranking function for a given dataset:

```
def getrankfunction(dataset):
  def rankfunction(population):
    scores=[(scorefunction(t,dataset),t) for t in population]
    scores.sort( )
    return scores
  return rankfunction
```

You're ready to automatically create a program that represents the formula for your mathematical dataset. Try this in your Python session:

```
>>> reload(gp)
>>> rf=gp.getrankfunction(gp.buildhiddenset( ))
>>> gp.evolve(2,500,rf,mutationrate=0.2,breedingrate=0.1,pexp=0.7,pnew=0.1)
16749
10674
5429
3090
491
151
151
0
add
 multiply
  p0
  add
   2
   p0
 add
  add
   p0
   4
  add
   p1
   add
    p1
    isgreater
     10
     5
```

The numbers change slowly, but they should decrease until they finally reach 0. Interestingly, the solution shown here gets everything correct, but it's quite a bit more complicated than the function used to create the dataset. (It's very likely that the solution you generated will also seem more complicated than it has to be.) However, a little algebra shows us that these functions are actually the same—remember that p0 is X and p1 is Y. The first line is the function represented by this tree:

```
 (X*(2+X))+X+4+Y+Y+(10>5)
= 2*X+X*X+X+4+Y+Y+1
= X**2 + 3*X + 2*Y + 5
```

This demonstrates an important property of genetic programming: the solutions it finds may well be correct or very good, but because of the way they are constructed, they will often be far more complicated than anything a human programmer would design. There will often be large sections of a program that don't do anything or that represent a complicated formula that returns the same value every time. Notice in the above example that the node `(10>5)` is just an odd way of saying 1.

It is possible to force the programs to remain simple, but in many cases this will make it more difficult to find a good solution. A better way to deal with this issue is to allow the programs to evolve to a good solution and then remove and simplify unnecessary portions of the tree. You can do this manually, and in some cases you can do it automatically using a pruning algorithm.

The Importance of Diversity

Part of the `evolve` function ranks the programs from best to worst, so it's tempting to just take two or three of the programs at the top and replicate and modify them to become the new population. After all, why would you bother allowing anything less than the best to continue?

The problem is that choosing only a couple of the top solutions quickly makes the population extremely homogeneous (or inbred, if you like), containing solutions that are all pretty good but that won't change much because crossover operations between them lead to more of the same. This problem is called reaching a *local maxima*, a state that is good but not quite good enough, and one in which small changes don't improve the result.

It turns out that having the very best solutions combined with a large number of moderately good solutions tends to lead to better results. For this reason, the `evolve` function has a couple of extra parameters that allow you to tune that amount of diversity in the selection process. By lowering the `probexp` value, you allow weaker solutions into the final result, turning the process from "survival of the fittest" to "survival of the fittest and luckiest." By increasing the `probnew` value, you allow completely new programs to be added to the mix occasionally. Both of these values increase the amount of diversity in the evolution process but won't disrupt it too much, since the very worst programs will always be eliminated eventually.

A Simple Game

A more interesting problem for genetic programming is building an AI for a game. You can force the programs to evolve by having them compete against each other and against real people, and giving the ones that win the most a higher chance of making it to the next generation. In this section, you'll create a simulator for a very simple game called Grid War, which is depicted in Figure 11-6.

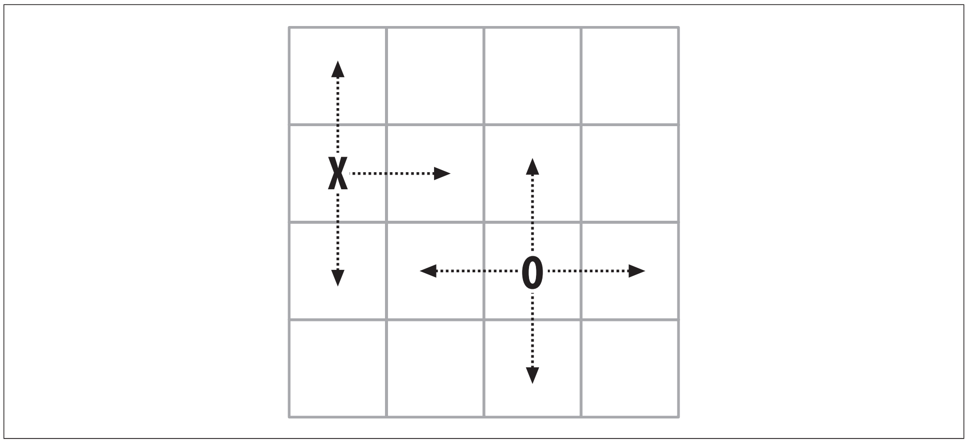


Figure 11-6. Grid War example

The game has two players who take turns moving around on a small grid. Each player can move in one of four directions, and the board is limited so if a player attempts to move off one side, he forfeits his turn. The object of the game is to capture the other player by moving onto the same square as his on your turn. The only additional constraint is that you automatically lose if you try to move in the same direction twice in a row. This game is very basic but since it pits two players against each other, it will let you explore more competitive aspects of evolution.

The first step is to create a function that uses two players and simulates a game between them. The function passes the location of the player and the opponent to each program in turn, along with the last move made by the player, and takes the return value as the move.

The move should be a number from 0 to 3, indicating one of four possible directions, but since these are random programs that can return any integer, the function has to handle values outside this range. To do this, it uses *modulo 4* on the result. Random programs are also liable to do things like create a player that moves in a circle, so the number of moves is limited to 50 before a tie is declared.

Add gridgame to *gp.py*:

```
def gridgame(p):
  # Board size
  max=(3,3)

  # Remember the last move for each player
  lastmove=[-1,-1]

  # Remember the player's locations
  location=[[randint(0,max[0]),randint(0,max[1])]]

  # Put the second player a sufficient distance from the first
  location.append([(location[0][0]+2)%4,(location[0][1]+2)%4])
```

```
    # Maximum of 50 moves before a tie
    for o in range(50):

      # For each player
      for i in range(2):
        locs=location[i][:]+location[1-i][:]
        locs.append(lastmove[i])
        move=p[i].evaluate(locs)%4

        # You lose if you move the same direction twice in a row
        if lastmove[i]==move: return 1-i
        lastmove[i]=move
        if move==0:
          location[i][0]-=1
          # Board limits
          if location[i][0]<0: location[i][0]=0
        if move==1:
          location[i][0]+=1
          if location[i][0]>max[0]: location[i][0]=max[0]
        if move==2:
          location[i][1]-=1
          if location[i][1]<0: location[i][1]=0
        if move==3:
          location[i][1]+=1
          if location[i][1]>max[1]: location[i][1]=max[1]

        # If you have captured the other player, you win
        if location[i]==location[1-i]: return i
    return -1
```

The program will return 0 if player 1 is the winner, 1 if player 2 is the winner, and –1 in the event of a tie. You can try building a couple of random programs and having them compete:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> p1=gp.makerandomtree(5)
>>> p2=gp.makerandomtree(5)
>>> gp.gridgame([p1,p2])
1
```

These programs are totally unevolved, so they probably lose by moving in the same direction twice in a row. Ideally, an evolved program will learn not to do this.

A Round-Robin Tournament

In keeping with collective intelligence, you would want the programs to test their fitness by playing against real people, and force their evolution that way. This would be a great way to capture the behavior of thousands of people and use it to develop a more intelligent program. However, with a large population and many generations, this could quickly add up to tens of thousands of games, and most of them would be against very poor opponents. That's impractical for our purposes, so you can first have the programs evolve by competing against each other in a tournament.

The `tournament` function takes a list of players as its input and pits each one against every other one, tracking how many times each program loses its game. Programs get two points if they lose and one point if they tie. Add `tournament` to *gp.py*:

```
def tournament(pl):
  # Count losses
  losses=[0 for p in pl]

  # Every player plays every other player
  for i in range(len(pl)):
    for j in range(len(pl)):
      if i==j: continue

      # Who is the winner?
      winner=gridgame([pl[i],pl[j]])

      # Two points for a loss, one point for a tie
      if winner==0:
        losses[j]+=2
      elif winner==1:
        losses[i]+=2
      elif winner==-1:
        losses[i]+=1
        losses[i]+=1
        pass

  # Sort and return the results
  z=zip(losses,pl)
  z.sort( )
  return z
```

At the end of the function, the results are sorted and returned with the programs that have the fewest losses at the top. This is the return type needed by `evolve` to evaluate programs, which means that `tournament` can be used as an argument to `evolve` and that you're now ready to evolve a program to play the game. Try it in your Python session (this may take some time):

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> winner=gp.evolve(5,100,gp.tournament,maxgen=50)
```

As the programs evolve, notice that the loss numbers don't strictly decrease like they did with the mathematical function. Take a minute to think about why this is—after all, the best player is always allowed into the next generation, right? As it turns out, since the next generation consists entirely of newly evolved programs, the best program in one generation might fare a lot worse in the next.

Playing Against Real People

Once you've evolved a program that performs well against its robotic competitors, it's time to battle against it yourself. To do this, you can create another class that also has an evaluate method that displays the board to the user and asks what move they want to make. Add the humanplayer class to *gp.py*:

```
class humanplayer:
  def evaluate(self,board):

    # Get my location and the location of other players
    me=tuple(board[0:2])
    others=[tuple(board[x:x+2]) for x in range(2,len(board)-1,2)]

    # Display the board
    for i in range(4):
      for j in range(4):
        if (i,j)==me:
          print 'O',
        elif (i,j) in others:
          print 'X',
        else:
          print '.',
      print

    # Show moves, for reference
    print 'Your last move was %d' % board[len(board)-1]
    print ' 0'
    print '2 3'
    print ' 1'
    print 'Enter move: ',

    # Return whatever the user enters
    move=int(raw_input( ))
    return move
```

In your Python session, you can take on your creation:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> gp.gridgame([winner,gp.humanplayer( )])
. O . .
. . . .
. . . .
. . . X
Your last move was -1
 0
2 3
 1
Enter move:
```

Depending on how well your program evolved, you may find it easy or difficult to beat. Your program will almost certainly have learned that it can't make the same move twice in a row, since that leads to instant death, but the extent to which it has mastered other strategies will vary with each run of `evolve`.

Further Possibilities

This chapter is just an introduction to genetic programming, which is a huge and rapidly advancing field. You've used it so far to approach simple problems in which programs are built in minutes rather than days, but the principles can be extended to much more complex problems. The number of programs in the populations here have been very small compared to those used in more complex problems—a population of thousands or tens of thousands is more typical. You are encouraged to come up with more difficult problems and try larger population sizes, but you may have to wait hours or days while the programs run.

The following section outlines a few ways in which the simple genetic programming model can be extended for different applications.

More Numerical Functions

We have used a very small set of functions to construct the programs so far. This limits the scope of what a simple program can do—for more complicated problems, it's necessary to greatly increase the number of functions available to build a tree. Here are some possible functions to add:

- Trigonometric functions like sine, cosine, and tangent
- Other mathematical functions like power, square root, and absolute value
- Statistical distributions, such as a Gaussian
- Distance metrics, like Euclidean and Tanimoto distances
- A three-parameter function that returns 1 if the first parameter is between the second and third
- A three-parameter function that returns 1 if the difference between the first two parameters is less than the third

These can get as complicated as you like, and they are often tailored to specific problems. Trigonometric functions may be a necessity when working in a field like signal processing, but they are not much use in a game like the one you built in this chapter.

Memory

The programs in this chapter are almost entirely reactive; they give a result based solely on their inputs. This is the right approach for solving mathematical functions, but it doesn't allow the programs to work from a longer-term strategy. The chasing game passes the programs the last move they made—mostly so the programs learn they can't make the same move twice in a row—but this is simply the output of the program, not something they set themselves.

For a program to develop a longer-term strategy, it needs a way to store information for use in the next round. One simple way to do this is to create new kinds of nodes that can store and retrieve values from predefined slots. A *store* node has a single child and an index of a memory slot; it gets the result from its child and stores it in the memory slot and then passes this along to its parent. A *recall* node has no children and simply returns the value in the appropriate slot. If a store node is at the top of the tree, the final result is available to any part of the tree that has the appropriate recall node.

In addition to individual memory, it's also possible to set up shared memory that can be read and written to by all the different programs. This is similar to individual memory, except that there are a set of slots that all the programs can read from and write to, creating the potential for higher levels of cooperation and competition.

Different Datatypes

The framework described in this chapter is for programs that take integer parameters and return integers as results. It can easily be altered to work with float values, since the operations are the same. To do this, simply alter `makerandomtree` to create the constant nodes with a random float value instead of a random integer.

Building programs that handle other kinds of data will require more extensive modification, mostly changing the functions on the nodes. The basic framework can be altered to handle types such as:

Strings
These would have operations like concatenate, split, indexing, and substrings.

Lists
These would have operations similar to strings.

Dictionaries
These would include operations like replacement and addition.

Objects
Any custom object could be used as an input to a tree, with the functions on the nodes being method calls to the object.

An important point that arises from these examples is that, in many cases, you'll require the nodes in the tree to process more than one type of return value. A substring operation, for example, requires a string and two integers, which means that one of its children would have to return a string and the other two would have to return integers.

The naïve approach to this would be to randomly generate, mutate, and breed trees, simply discarding the ones in which there is a mismatch in datatypes. However, this would be computationally wasteful, and you've already seen how you can put a constraint on the way trees are constructed—every function in the integer trees knows how many children it needs, and this can be easily extended to constrain the types of children and their return types. For example, you might redefine the fwrapper class like the following, where params is a list of strings specifying datatypes that can be used for each parameter:

```
class fwrapper:
  def __init__(self,function,params,name):
    self.function=function
    self.childcount=param
    self.name=name
```

You'd also probably want to set up flist as a dictionary with return types. For example:

```
flist={'str':[substringw,concatw],'int':[indexw,addw,subw]}
```

Then you could change the start of makerandomtree to something like:

```
def makerandomtree(pc,datatype,maxdepth=4,fpr=0.5,ppr=0.5):
  if random( )<fpr and maxdepth>0:
    f=choice(flist[datatype])
    # Call makerandomtree with all the parameter types for f
    children=[makerandomtree(pc,type,maxdepth-1,fpr,ppr)
              for type in f.params]
    return node(f,children)
etc...
```

The crossover function would also have to be altered to ensure that swapped nodes have the same return type.

Ideally, this section has given you some ideas about how genetic programming can be extended from the simple model described here, and has inspired you to improve it and to try automatically generating programs for more complex problems. Although they may take a very long time to generate, once you find a good program, you can use it again and again.

Exercises

1. *More function types*. We started with a very short list of functions. What other functions can you think of? Implement a Euclidean distance node with four parameters.
2. *Replacement mutation*. Implement a mutation procedure that chooses a random node on the tree and changes it. Make sure it deals with function, constant, and parameter nodes. How is evolution affected by using this function instead of the branch replacement?
3. *Random crossover*. The current crossover function chooses branches from two trees at the same level. Write a different crossover function that crosses any two random branches. How does this affect evolution?
4. *Stopping evolution*. Add an additional criteria to `evolve` that stops the process and returns the best result if the best score hasn't improved within X generations.
5. *Hidden functions*. Try creating other mathematical functions for the programs to guess. What sort of functions can be found easily, and which are more difficult?
6. *Grid War player*. Try to hand-design your own tree program that does well at Grid War. If you find this easy, try to write another completely different one. Instead of having a completely random initial population, make it *mostly* random, with your hand-designed programs included. How do they compare to random programs, and can they be improved with evolution?
7. *Tic-tac-toe*. Build a tic-tac-toe simulator for your programs to play. Set up a tournament similar to the Grid War tournament. How well do the programs do? Can they ever learn to play perfectly?
8. *Nodes with datatypes*. Some ideas were provided in this chapter about implementing nodes with mixed datatypes. Implement this and see if you can evolve a program that learns to return the second, third, sixth, and seventh characters of a string (e.g., "genetic" becomes "enic").

Selected Chapters

From:

Grokking

Deep Learning

by Andrew W. Trask

Introduction to Neural Prediction
Forward Propagation

3

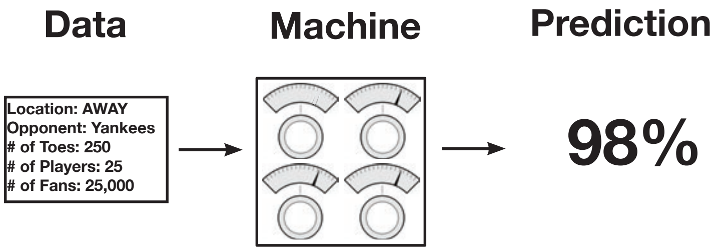
IN THIS CHAPTER

> I try not to get involved in the business of prediction. It's a quick way to look like an idiot.
>
> — *WARREN ELLIS*

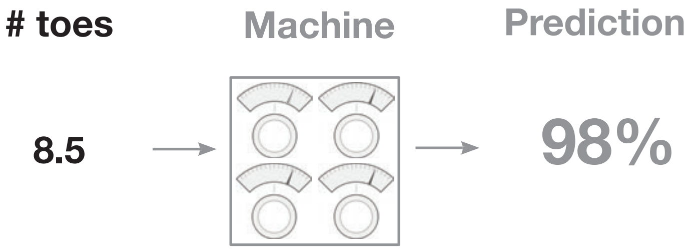
Step 1: Predict

This chapter is about "Prediction"

In the previous chapter, we learned about the paradigm: "Predict, Compare, Learn". In this chapter, we will dive deep into the first step, "Predict". You may remember that the predict step looks a lot like this.

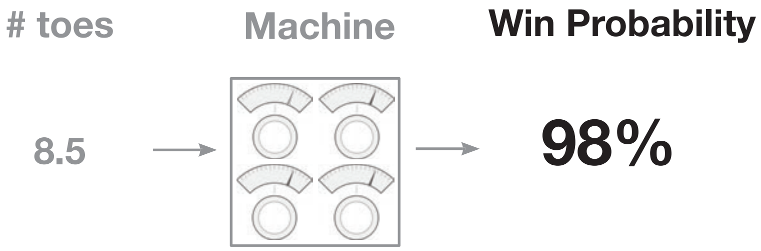


In this chapter, we're going to learn more about what these 3 different parts of a neural network prediction really look like under the hood. Let's start with the first one, the Data. In our first neural network, we're going to predict one datapoint at a time, like so.

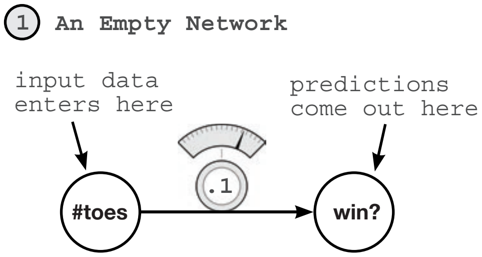


Later on, we will find that the "number of datapoints at a time" that we want to process will have a significant impact on what our network looks like. You might be wondering, "how do I choose how many datapoints to propagate at a time?" The answer to this question is based on whether or not you think the neural network can be accurate with the data you give it. For example, if I'm trying to predict whether or not there's a cat in a photo, I definitely need to show my network all the pixels of an image at once. Why? Well, if I only sent you one pixel of an image, could you classify whether the image contained a cat? Me neither! (That's a general rule of thumb by the way. Always present enough information to the network, where "enough information" is defined loosely as how much a human might need to make the same prediction).

Let's skip over the network for now. As it turns out, we can only create our network once we understand the shape of our input and output datasets (For now, shape means "number of columns" or "number of datapoints we're processing at once"). For now, we're going to stick with the "single-prediction" of "likelihood that the baseball team will win".



Ok, so now that we know that we want to take one input datapoint and output one prediction, we can create our neural network. Since we only have one input datapoint and one output datapoint, we're going to build a network with a single knob mapping from the input point to the output. Abstractly these "knob"s are actually called "weight"s, and we will refer to them as such from here on out. So, without further ado, here's our first neural network with a single weight mapping from our input "#toes" to output "win?"

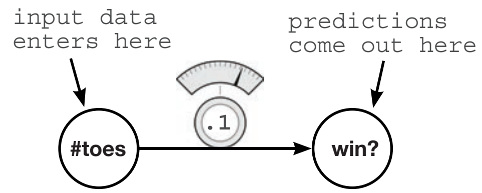


As you can see, with one weight, this network takes in one datapoint at a time (average number of toes on the baseball team) and outputs a single prediction (whether or not it thinks the team will win).

A Simple Neural Network Making a Prediction

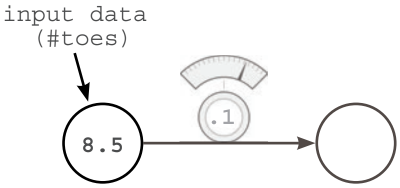
Let's start with the simplest neural network possible.

1 An Empty Network



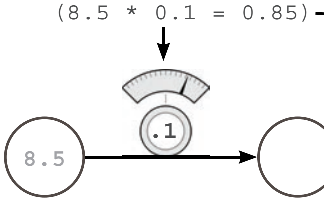
```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

2 Inserting One Input Datapoint



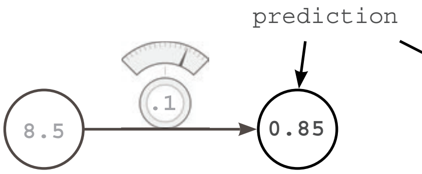
```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)

print(pred)
```

3 Multiplying Input By Weight



```
def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

4 Depositing Prediction



```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
```

What is a Neural Network?

This is a neural network.

Open up a Jupyter Notebook and run the following:

the network

```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

how we use the network to predict something

```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
print(pred)
```

You just made your first neural network and used it to predict! Congratulations! The last line prints the prediction (`pred`). It should be 0.85. So what is a neural network? For now, it's one or more *weights* which we can multiply by our `input` data to make a `prediction`.

What is `input` data?
It's a number that we recorded in the real world somewhere. It's usually something that is easily knowable, like today's temperature, a baseball player's batting average, or yesterday's stock price.

What is a `prediction`?
A `prediction` is what the neural network tells us *given our input data* such as "given the temperature, it is **0%** likely that people will wear sweatsuits today" or "given a baseball player's batting average, he is **30%** likely to hit a home run" or "given yesterday's stock price, today's stock price will be **101.52**".

Is this `prediction` always right?
No. Sometimes our neural network will make mistakes, but it can learn from them. For example, if it predicts too high, it will adjust it's `weight` to predict lower next time and vice versa.

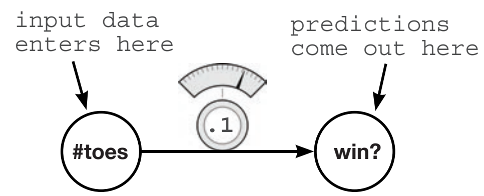
How does the network learn?
Trial and error! First, it tries to make a `prediction`. Then, it sees whether it was too high or too low. Finally, it changes the `weight` (up or down) to predict more accurately the next time it sees the same `input`.

What does this Neural Network do?

It multiplies the `input` by a `weight`. It "scales" the input by a certain amount.

On the previous page, we made our first prediction with a neural network. A neural network, in it's simplest form, uses the power of *multiplication*. It takes our input datapoint (in this case, 8.5) and *multiplies* it by our `weight`. If the `weight` is 2, then it would *double our input*. If the `weight` is 0.01, then it would *divide* the input by 100. As you can see, some `weight` values make the input *bigger* and other values make it *smaller*.

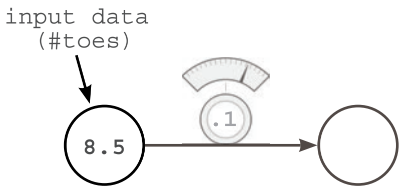
1 **An Empty Network**



```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

The interface for our neural network is really quite simple. It accepts an `input` variable as *information*, and a weight variable as *knowledge* and outputs a `prediction`. Every neural network you will ever see works this way. It uses the *knowledge* in the weights to interpret the *information* in the input data. Later neural networks will accept larger, more complicated `input` and `weight` values, but this same underlying premise will always ring true.

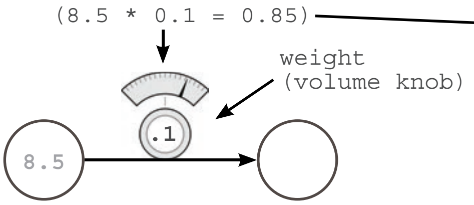
2 **Inserting One Input Datapoint**



```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
```

In this case, the "information" is the average number of toes on a baseball team before a game. Notice several things. The neural network does NOT have access to any information except *one* instance. If, after this prediction, we were to feed in number_of_toes[1], it would not remember the prediction it made in the last timestep. A neural network only knows what you feed it as input. It forgets everything else. Later, we will learn how to give neural networks "short term memories" by feeding in multiple inputs at once.

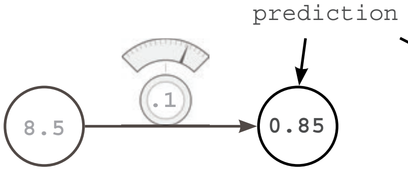
3 Multiplying Input By Weight



```
def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

Another way to think about a neural network's `weight` is as a measure of *sensitivity* between the input of the network and its prediction. If the weight is very high, then even the tiniest input can create a really large prediction! If the weight is very small, then even large inputs will make small predictions. This *sensitivity* is very akin to ***volume***. "Turning up the weight" amplifies our prediction relative to our input. `weight` is a volume knob!

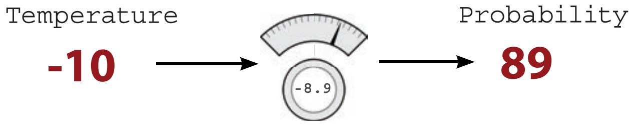
4 Depositing Prediction



```
number_of_toes = [8.5, 9.5, 10, 9]
input = number_of_toes[0]
pred = neural_network(input,weight)
```

So in this case, what our neural network is really doing is applying a *volume knob* to our `number_of_toes` variable. In theory, this *volume knob* is able to tell us the likelihood that the team will win based on the average number of toes per player on our team. And this may or may not work. Truthfully, if the team had 0 toes, they would probably play terribly. However, baseball is much more complex than this. On the next page, we will present multiple pieces of information at the same time, so that the neural network can make more informed decisions.

Before we go, neural networks don't just predict positive numbers either, they can also *predict negative numbers*, and even take *negative numbers as input*. Perhaps you want to predict the "probability that people will wear coats today", if the temperature was -10 degrees Celsius, then a negative weight would predict a high probability that people would wear coats today.

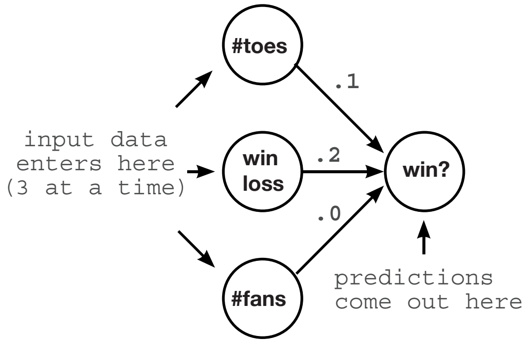


Making a Prediction with Multiple Inputs

Neural Networks can combine intelligence from multiple datapoints.

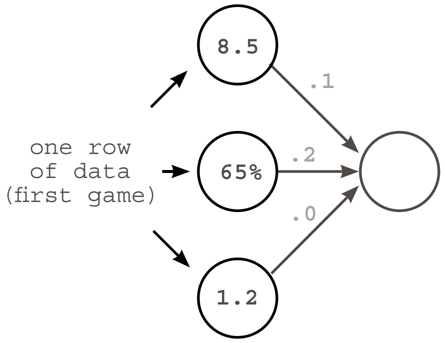
Our last neural network was able to take one datapoint as input and make one prediction based on that datapoint. Perhaps you've been wondering, "is average # of toes really a very good predictor?... all by itself?" If so, you're onto something. What if we were able to give our network more information (at one time) than just the "average number of toes". It should, in theory, be able to make more accurate predictions, yes? Well, as it turns out, our network can accept multiple input datapoints at a time. See the prediction below!

1 **An Empty Network With Multiple Inputs**



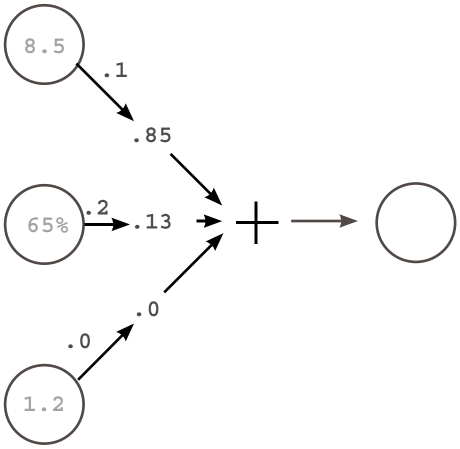
```
weights = [0.1, 0.2, 0]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

2 **Inserting One Input Datapoint**



```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */


toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

3 Perform a Weighted Sum of Inputs

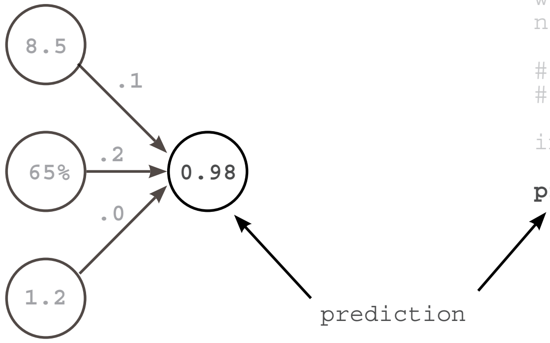


```
def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred


def w_sum(a,b):

    assert(len(a) == len(b))

    output = 0

    for i in range(a):
       output += (a[i] * b[i])

    return output
```

```
                            local
  inputs   weights   predictions
 ( 8.50  *  0.1 )  =  0.85  = toes prediction
 ( 0.65  *  0.2 )  =  0.13  = wlrec prediction
 ( 1.20  *  0.0 )  =  0.00  = fans prediction

toes prediction + wlrec prediction + fans prediction = final prediction

      0.85      +       0.13       +       0.00      =     0.98
```

4 Deposit Prediction

```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

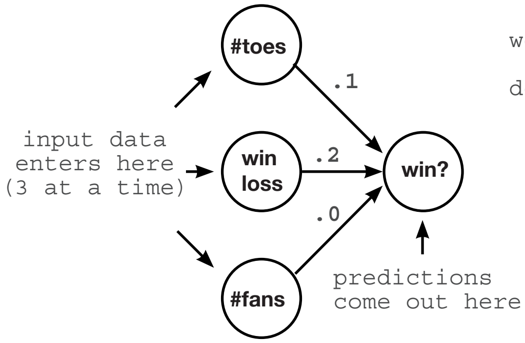


Multiple Inputs - What does this Neural Network do?

It multiplies 3 `inputs` by 3 `knob_weights` and sums them. This is a "weighted sum".

At the end of the previous section, we came to realize the limiting factor of our simple neural network, it is only a volume knob on one datapoint. In our example, that datapoint was the average number of toes on a baseball team. We realized that in order to make accurate predictions, we need to build neural networks that can *combine multiple inputs at the same time*. Fortunately, neural networks are perfectly capable of doing so.

1 **An Empty Network With Multiple Inputs**



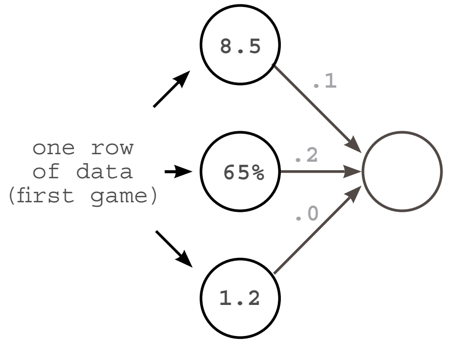
```
weights = [0.1, 0.2, 0]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

In this new neural network, we can accept *multiple inputs at a time* per prediction. This allows our network to combine various forms of information to make more well informed decisions. However, the fundamental mechanism for using our weights has not changed. We still take each input and run it through its own volume knob. In other words, we take each input and multiply it by its own weight. The new property here is that, since we have mutliple inputs, we have to sum their respective predictions. Thus, we take each input, multiply it by its respective weight, and then sum all the local predictions together. This is called a "weighted sum of the input" or a "weighted sum" for short. Some also refer to this "weighted sum" as a "dot product" as we'll see.

A Relevant Reminder

The interface for our neural network is quite simple. It accepts an input variable as *information*, and a weight variable as *knowledge* and outputs a prediction.

2 Inserting One Input Datapoint



```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */


toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

This new need to process multiple inputs at a time justifies the use of a new tool. This tool is called a **vector** and if you've been following along in your iPython notebook, you've already been using it. A vector is nothing other than a *list of numbers.* `input` is a vector and `weights` is a vector. Can you spot any more vectors in the code above (there are 3 more)?

As it turns out, vectors are incredibly useful whenever you want to perform operations involving groups of numbers. In this case, we're performing a weighted sum between two vectors (dot product). We're taking two vectors of equal length (input and weights), multiplying each number based on its position (the first position in input is multiplied by the first position in weights, etc.), and then summing the resulting output.

It turns out that whenever we perform a mathematical operation between two vectors of equal length where we "pair up" values according to their position in the vector (again... position 0 with 0, 1, with 1, and so on), we call this an **elementwise** operation. Thus "elementwise addition" sums two vectors. "elementwise multiplication" multiplies two vectors.

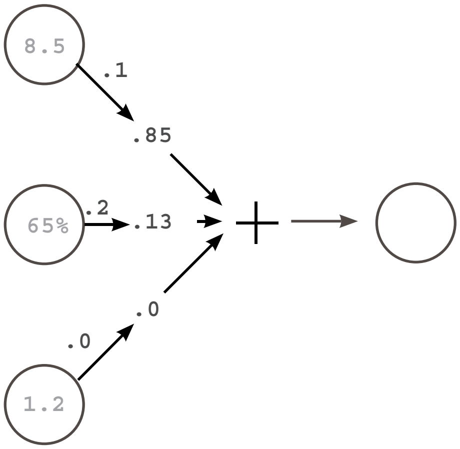
Challenge: Vector Math

Being able to manipulate vectors is a cornerstone technique for Deep Learning. See if you can write functions that perform the following operations:

def elementwise_multiplication(vec_a, vec_b)
def elementwise_addition(vec_a, vec_b)
def vector_sum(vec_a)
def vector_average(vec_a)

Then, see if you can use two of these methods to perform a dot product!

3 Perform a Weighted Sum of Inputs



```
def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred


def w_sum(a,b):

    assert(len(a) == len(b))

    output = 0

    for i in range(a):
       output += (a[i] * b[i])

    return output
```

```
                        local
   inputs   weights   predictions
  ( 8.50  *  0.1 )  =  0.85  = toes prediction
  ( 0.65  *  0.2 )  =  0.13  = wlrec prediction
  ( 1.20  *  0.0 )  =  0.00  = fans prediction

toes prediction + wlrec prediction + fans prediction = final prediction

      0.85      +       0.13       +       0.00      =     0.98
```

The intuition behind how and why a dot product (weighted sum) works is easily one of the most important parts of truly understanding how neural networks make predictions. Loosely stated, a dot product gives us a *notion of similarity* between two vectors. Consider the examples:

```
                             w_sum(a,b)  = 0
a = [ 0, 1, 0, 1]            w_sum(b,c)  = 1
b = [ 1, 0, 1, 0]            w_sum(b,d)  = 1
c = [ 0, 1, 1, 0]            w_sum(c,c)  = 2
d = [.5, 0,.5, 0]            w_sum(d,d)  = .5
e = [ 0, 1,-1, 0]            w_sum(c,e)  = 0
```

The highest weighted sum (w_sum(c,c)) is between vectors that are exactly identical. In contrast, since a and b have no overlapping weight, their dot product is zero. Perhaps the most interesting weighted sum is between c and e, since e has a negative weight. This negative weight cancelled out the positive similarity between them. However, a dot product between e and itself would yield the number 2, despite the negative weight (double negative turns positive). Let's become familiar with these properties.

Some have equated the properties of the "dot product" to a "logical AND". Consider a and b.

```
a = [ 0, 1, 0, 1]
b = [ 1, 0, 1, 0]
```

If you asked whether both a[0] AND b[0] had value, the answer would be no. If you asked whether both a[1] AND b[1] had value, the answer would again be no. Since this is ALWAYS true for all 4 values, the final score equals 0. Each value failed the logical AND.

```
b = [ 1, 0, 1, 0]
c = [ 0, 1, 1, 0]
```

b and c, however, have one column that shares value. It passes the logical AND since b[2] AND c[2] have weight. This column (and only this column) causes the score to rise to 1.

```
c = [ 0, 1, 1, 0]
d = [.5, 0,.5, 0]
```

Fortunately, neural networks are also able to model partial ANDing. In this case, c and d share the same column as b and c, but since d only has 0.5 weight there, the final score is only 0.5. We exploit this property when modeling probabilities in neural networks.

```
d = [.5, 0,.5, 0]
e = [-1, 1, 0, 0]
```

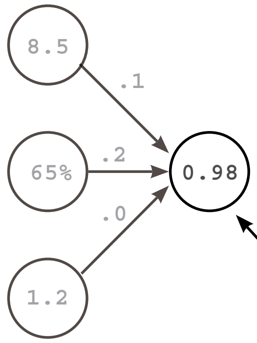
In this analogy, negative weights tend to imply a logcal NOT operator, given that any positive weight paired with a negative weight will cause the score to go down. Furthermore, if both vectors have negative weights (such as w_sum(e,e)), then it will perform a *double negative* and add weight instead. Additionally, some will say that it's an OR after the AND, since if any of the rows show weight, the score is affected. Thus, for w_sum(a,b), if (a[0] AND b[0]) OR (a[1] AND b[1)...etc.. then have a positive score. Furthermore, if one is negative, then that column gets a NOT. Amusingly, this actually gives us a kind of crude language to "read our weights". Let's "read" a few examples, shall we? These assume you're performing w_sum(input,weights) and the "then" to these "if statements" is just an abstract "then give high score".

```
weights = [ 1, 0, 1] => if input[0] OR input[2]

weights = [ 0, 0, 1] => if input[2]

weights = [ 1, 0, -1] => if input[0] OR NOT input[2]

weights = [ -1, 0, -1] => if NOT input[0] OR NOT input[2]

weights = [ 0.5, 0, 1] => if BIG input[0] or input[2]
```

Notice in the last row that a weight[0] = 0.5 means that the corresponding input [0] would have to be larger to compensate for the smaller weighting. And as I mentioned, this is a very *very* crude approximate language. However, I find it to be immensely useful when trying to picture in my head what's going on under the hood. This will help us significantly in the future, especially when putting networks together in increasingly complex ways.

So, given these intuitions, what does this mean when our neural network makes a prediction? Very rougly speaking, it means that our network gives a high score of our inputs based on *how similar they are to our weights*. Notice below that "nfans" is completely ignored in the prediction because the weight associated with it is a 0. The most sensitive predictor, in fact, is "wlrec" because its weight is a 0.2. However, the dominant force in the high score is the number of toes ("ntoes") not because the weight is the highest, but because the input combined with the weight is by far the highest.

4 Deposit Prediction



```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

prediction

A few more points that we will note here for further reference. We cannot shuffle our weights. They have specific positions they need to be in. Furthermore, both the value of the weight AND the value of the input determine the overall impact on the final score. Finally, a negative weight would cause some inputs to reduce the final prediction (and vise versa).

Multiple Inputs - Complete Runnable Code

The code snippets from this example come together as follows.

We can create and execute our neural network using the following code. For the purposes of clarity, I have written everything out using only basic properties of Python (lists and numbers). However, there is a better way that we will start using in the future.

There is a python library called "numpy" which stands for "numerical python". It has very efficient code for creating vectors and performing common functions (such as a dot product). So, without further ado, here's the same code in numpy.

Numpy Code

```
import numpy as np

weights = np.array([0.1, 0.2, 0])

def neural_network(input, weights):

    pred = input.dot(weights)

    return pred


toes =  np.array([8.5, 9.5, 9.9, 9.0])
wlrec = np.array([0.65, 0.8, 0.8, 0.9])
nfans = np.array([1.2, 1.3, 0.5, 1.0])

# input corresponds to every entry
# for the first game of the season

input = np.array([toes[0],wlrec[0],nfans[0]])
pred = neural_network(input,weight)
print(pred)
```

Previous Code

```
def w_sum(a,b):

    assert(len(a) == len(b))

    output = 0

    for i in range(a):
       output += (a[i] * b[i])

    return output

weights = [0.1, 0.2, 0]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]
pred = neural_network(input,weight)
print(pred)
```

Both networks should simply print out:

0.98

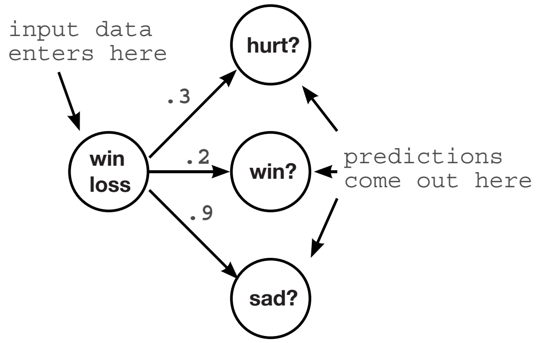
Notice that we didn't have to create a special "w_sum" function. Instead, numpy has a special function called "dot" (short for "dot product") which we can call. Many of the functions we want to use in the future will have numpy parallels, as we will see later.

Making a Prediction with Multiple Outputs

Neural Networks can also make multiple predictions using only a single input.

Perhaps a simpler augmentation than multiple inputs is multiple outputs. Prediction occurs in the same way as if there were 3 disconnected single-weight neural networks.

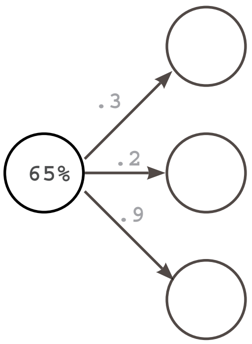
1 An Empty Network With Multiple Outputs



```
/* instead of predicting just
whether the team won or lost,
now we're also predicting whether
they are happy/sad AND the percentage
of the team that is hurt. We are
making this prediction using only
the current win/loss record */

weights = [0.3, 0.2, 0.9]

def neural_network(input, weights):

    pred = ele_mul(input,weights)

    return pred
```

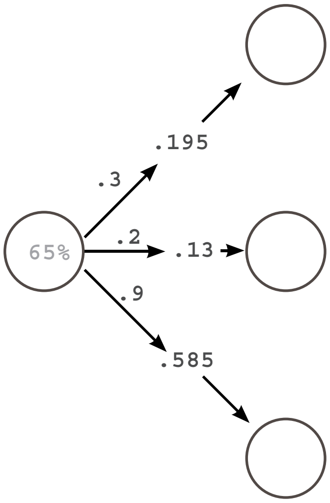
The most important commentary in this setting is to notice that the 3 predictions really are completely separate. Unlike neural networks with multiple inputs and a single output where the prediction is undeniably connected this network truly behaves as 3 independent components, each receiving the same input data. This makes the network quite trivial to implement.

2 Inserting One Input Datapoint



```
wlrec = [0.65, 0.8, 0.8, 0.9]

input = wlrec[0]

pred = neural_network(input,weight)
```

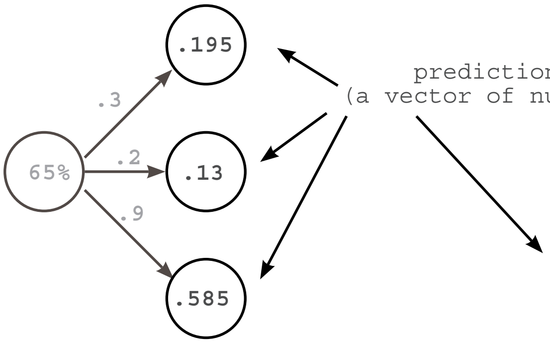
3 Perform an Elementwise Multiplication



```
def neural_network(input, weights):

   pred = ele_mul(input,weights)

   return pred


def ele_mul(number,vector):

   output = [0,0,0]

   assert(len(output) == len(vector))

   for i in xrange(len(vector)):
      output[i] = number * vector[i]

   return output
```

```
                          final
 inputs   weights      predictions
( 0.65  *  0.3 )  =  0.195  = hurt prediction
( 0.65  *  0.2 )  =  0.13   = win prediction
( 0.65  *  0.9 )  =  0.585  = sad prediction
```

4 Deposit Predictions



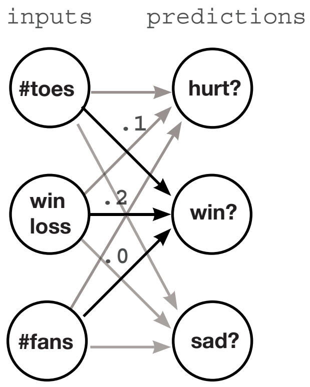
```
wlrec = [0.65, 0.8, 0.8, 0.9]

input = wlrec[0]

pred = neural_network(input,weight)
```

Predicting with Multiple Inputs & Outputs

Neural networks can predict multiple outputs given multiple inputs.

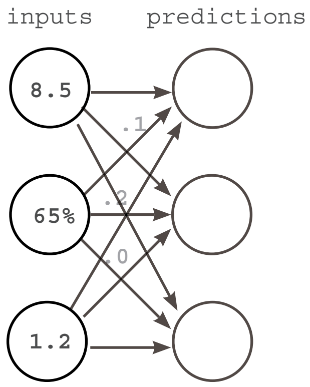
Finally, the way in which we built a network with multiple inputs or outputs can be combined together to build a network that has both multiple inputs AND multiple outputs. Just like before, we simply have a weight connecting each input node to each output node and prediction occurs in the usual way.

1 An Empty Network With Multiple Inputs & Outputs



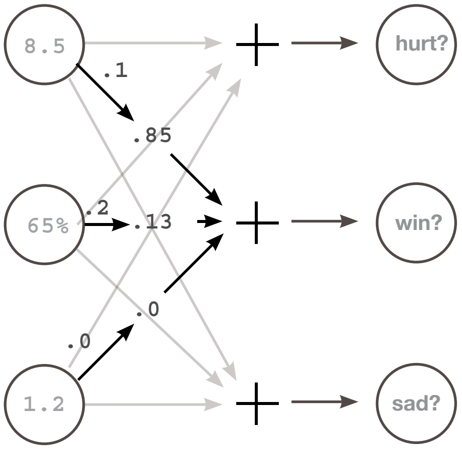
```
              #toes %win #fans
weights = [ [0.1, 0.1, -0.3],#hurt?
             [0.1, 0.2, 0.0], #win?
             [0.0, 1.3, 0.1] ]#sad?

def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred
```

2 Inserting One Input Datapoint



```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */


toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

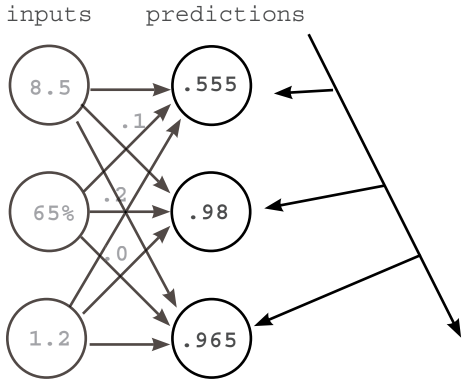
3 For Each Output, Perform a Weighted Sum of Inputs



```
def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred


def vect_mat_mul(vect,matrix):

    assert(len(a) == len(b))

    output = 0

    for i in range(a):
       output += (a[i] * b[i])

    return output
```

```
  #toes           %win          #fans

(8.5 * 0.1) + (0.65 * 0.1) + (1.2 * -0.3) = 0.555 = hurt prediction
(8.5 * 0.1) + (0.65 * 0.2) + (1.2 * 0.0)  = 0.98  = win prediction
(8.5 * 0.0) + (0.65 * 1.3) + (1.2 * 0.1)  = 0.965 = sad prediction
```

4 Deposit Predictions



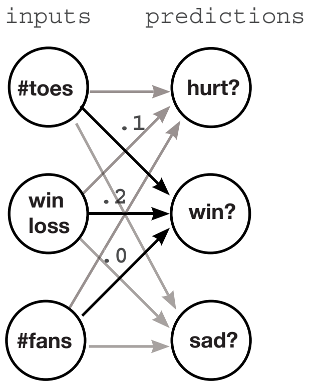
```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

Multiple Inputs & Outputs - How does it work?

It performs 3 independent weighted sums of the input to make 3 predictions.

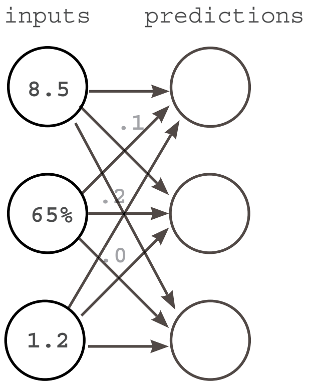
I find that there are 2 perspectives one can take on this architecture. You can either think of it as 3 weights coming out of each input node, or 3 weights going into each output node. For now, I find the latter to be much more beneficial. For now, think about this neural network as 3 independent dot products, 3 independent weighted sums of the input. Each output node takes its own weighted sum of the input and makes a prediction.

1 An Empty Network With Multiple Inputs & Outputs



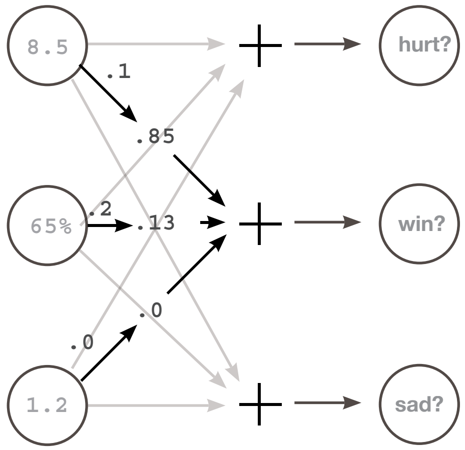
```
                #toes %win #fans
weights = [ [0.1, 0.1, -0.3],#hurt?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ]#sad?

def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred
```

2 Inserting One Input Datapoint



```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

3 For Each Output, Perform a Weighted Sum of Inputs



```
def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred

def vect_mat_mul(vect,matrix):

    assert(len(a) == len(b))

    output = vector_of_zeros(len(vect))

    for i in range(len(vect)):
       output[i] = w_sum(vect,matrix[i])

    return output
```

```
   #toes         %win         #fans

(8.5 * 0.1) + (0.65 * 0.1) + (1.2 * -0.3)  = 0.555 = hurt prediction
(8.5 * 0.1) + (0.65 * 0.2) + (1.2 * 0.0)   = 0.98  = win prediction
(8.5 * 0.0) + (0.65 * 1.3) + (1.2 * 0.1)   = 0.965 = sad prediction
```

As mentioned on the previous page, we are choosing to think about this network as a series of weighted sums. Thus, in the code above, we created a new function called "vect_mat_mul". This function iterates through each row of our weights (each row is a vector), and makes a prediction using our w_sum function. It is literally performing 3 consecutive weighted sums and then storing their predictions in a vector called "output". There's a lot more weights flying around in this one, but isn't that much more advanced than networks we have previously seen.

I want to use this "list of vectors" and "series of weighted sums" logic to introduce you to two new concepts. See the weights variable in step (1)? It's a list of vectors. A list of vectors is simply called a **matrix**. It is as simple as it sounds. Furthermore, there are functions that we will find ourselves commonly using that leverage matrices. One of these is called **vector-matrix multiplication**. Our "series of weighted sums" is exactly that. We take a vector, and perform a dot product with every row in a matrix**. As we will find out on the next page, we even have special numpy functions to help us out.

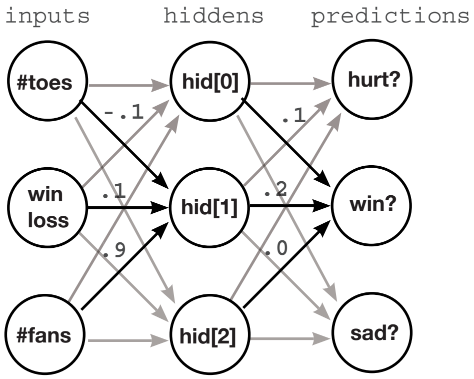
** Note: For those of you experienced with Linear Algebra, the more formal definition would store/process weights as column vectors instead of row vectors. This will be rectified shortly.

Predicting on Predictions

Neural networks can be stacked!

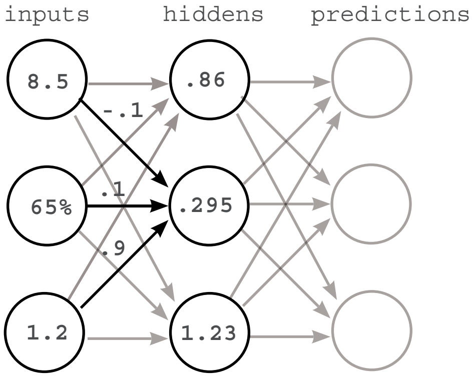
As the pictures below make clear, one can also take the output of one network and feed it as input to another network. This results in two consecutive vector-matrix multiplications. It may not yet be clear why you would predict in this way. However, some datasets (such as image classification) contain patterns that are simply too complex for a single weight matrix. Later, we will discuss the nature of these patterns. For now, it is sufficient that you know this is possible.

1 **An Empty Network With Multiple Inputs & Outputs**



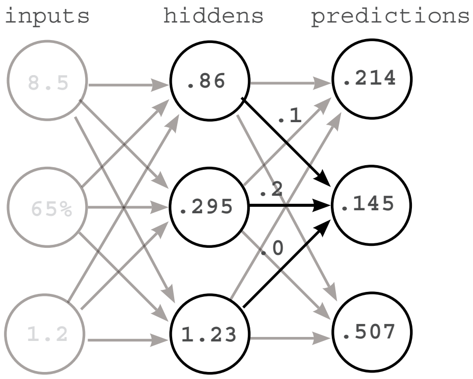
```
              #toes %win #fans
ih_wgt = [  [0.1, 0.2, -0.1],#hid[0]
            [-0.1,0.1, 0.9], #hid[1]
            [0.1, 0.4, 0.1] ]#hid[2]

         #   hid[0] hid[1] hid[2]
hp_wgt = [  [0.3, 1.1, -0.3],#hurt?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ]#sad?

weights = [ih_wgt, hp_wgt)

def neural_network(input, weights):

    hid = vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred
```

2 **Predicting the Hidden Layer**



```
    toes =  [8.5, 9.5, 9.9, 9.0]
    wlrec = [0.65,0.8, 0.8, 0.9]
    nfans = [1.2, 1.3, 0.5, 1.0]

    # input corresponds to every entry
    # for the first game of the season

    input = [toes[0],wlrec[0],nfans[0]]

    pred = neural_network(input,weight)


def neural_network(input, weights):

    hid = vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred
```

3 Predicting the Output Layer (and depositing the prediction)



```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)


def neural_network(input, weights):

    hid = vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred
```

Numpy Version

```
import numpy as np

#toes %win #fans
ih_wgt = np.array([
            [0.1, 0.2, -0.1],#hid[0]
            [-0.1,0.1, 0.9], #hid[1]
            [0.1, 0.4, 0.1]]).T #hid[2]


# hid[0] hid[1] hid[2]
hp_wgt = np.array([
            [0.3, 1.1, -0.3],#hurt?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ]).T#sad?

weights = [ih_wgt, hp_wgt]

def neural_network(input, weights):

    hid = input.dot(weights[0])
    pred = hid.dot(weights[1])
    return pred


toes =  np.array([8.5, 9.5, 9.9, 9.0])
wlrec = np.array([0.65,0.8, 0.8, 0.9])
nfans = np.array([1.2, 1.3, 0.5, 1.0])

input = np.array([toes[0],wlrec[0],nfans[0]])

pred = neural_network(input,weights)
print pred
```

A Quick Primer on Numpy

Numpy is so easy to use that it does a few things for you. Let's reveal the magic.

So far in this chapter, we've discussed two new types of mathematical tools, vectors and matrices. Furthermore, we have learned about different operations that occur on vectors and matrices including dot products, elementwise multiplication and addition, as well as vector-matrix multiplication. For these operations, we've written our own python functions that can operate on simple python "list" objects. In the short term, we will keep writing/using these functions so that we make sure we fully understand what's going on inside them. However, now that we've mentioned both "numpy" and several of the big operations, I'd like to give you a quick run-down of basic "numpy" use so that you will be ready for our transition to "only numpy" a few chapters from now. So, let's just start with the basics again, vectors and matrices.

```
import numpy as np

a = np.array([0,1,2,3]) # a vector
b = np.array([4,5,6,7]) # another vector
c = np.array([[0,1,2,3],# a matrix
              [4,5,6,7]])

d = np.zeros((2,4))#(2x4 matrix of zeros)
e = np.random.rand(2,5) # random 2x5
# matrix with all numbers between 0 and 1

print a
print b
print c
print d
print e
```

Output

```
[0 1 2 3]
[4 5 6 7]
[[0 1 2 3]
 [4 5 6 7]]
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
[[ 0.22717119  0.39712632
0.0627734   0.08431724
0.53469141]
 [ 0.09675954  0.99012254
0.45922775  0.3273326
0.28617742]]
```

We can create vectors and marices in multiple ways in numpy. Most of the common ones for neural networks are listed above. Note that the processes for creating a vector and a matrix are identical. If you create a matrix with only one row, you're creating a vector. Furthermore, as in mathematics in general, you create a matrix by listing (rows,columns). I say that only so that you can remember the order. Rows comes first. Columns comes second. Let's see some operations we can do on these vectors and matrices.

```
print a * 0.1 # multiplies every number in vector "a" by 0.1
print c * 0.2 # multiplies every number in matrix "c" by 0.2
print a * b # multiplies elementwise between a and b (columns paired up)
print a * b * 0.2 # elementwise multiplication then multiplied by 0.2
print a * c # since c has the same number of columns as a, this performs
# elementwise multiplication on every row of the matrix "c"

print a * e # since a and e don't have the same number of columns, this
# throws a "Value Error: operands could not be broadcast together with.."
```

Go ahead and run all of the code on the previous page. The first big of "at first confusing but eventually heavenly" magic should be visible on that page. When you multiply two variables with the "*" function, numpy automatically detects what kinds of variables you're working with and "tries" to figure out the operation you're talking about. This can be mega-convenient but sometimes makes numpy a bit hard to read. You have to make sure you keep up with what each variable type is in your head as you go along.

The general rule of thumb for anything elementwise (+,-,*,/) is that the two variables must either have the SAME number of columns, or one of the variables must only have 1 column.

For example, "print a * 0.1" takes a vector and multiplies it by a single number (a scalar). Numpy goes "oh, I bet I'm supposed to do vector-scalar multiplication here" and then it takes the scalar (0.1) and multiplies it by every value in the vector. This looks exactly the same as "print c * 0.2", except that numpy knows that c is a matrix. Thus, it performs scalar-matrix multiplication, multiplying every element in c by 0.2. Because the scalar has only one column, you can multiply it by anything (or divide, add, or subtract for that matter)

Next up, "print a * b". Numpy first identifies that they're both vectors. Since neither vector has only 1 column, it checks to see if they have an identical number of columns. Since they do, it knows to simply multiply each element by each element based on their positions in the vectors. The same is true with addition, subtraction and division.

"print a * c" is perhaps the most elusive. "a" is a vector with 4 columns. "c" is a (2x4) matrix. Neither have only one column, so numpy checks to see if they have the same number of columns. Since they do, numpy multiplies the vector "a" by each row of "c" (as if it was doing elementwise vector multiplication on each row).

Again, the most confusing part about this is that all of these operations look the same if you don't know which variables are scalars, vectors, or matrices. When I'm "reading numpy", I'm really doing 2 things, reading the operations and keeping track of the "shape" (number of rows and columns) of each operation. It'll take some practice, but eventually it becomes second nature.

```
a = np.zeros((1,4)) # vector of length 4
b = np.zeros((4,3)) # matrix with 4 rows & 3 columns

c = a.dot(b)
print c.shape
```

Output

```
(1,3)
```

There is one golden rule when using the 'dot' function. If you put the (rows,cols) description of the two variables you're "dotting" next to each other, neighboring numbers should always be the same. In this case, we're dot producting a (1,4) with a (4,3). Thus, it works fine, and outputs a (1,3). In terms of variable shape, you can think of it this way. Regardless of whether you're "dotting" vectors or matrices. Their "shape" (number of rows and columns) must line up. The columns on the "left" matrix must equal rows on the "right".

```
(a,b).dot(b,c) = (a,c)
```

```
a = np.zeros((2,4)) # matrix with 2 rows and 4 columns
b = np.zeros((4,3)) # matrix with 4 rows & 3 columns

c = a.dot(b)
print c.shape # outputs (2,3)


e = np.zeros((2,1)) # matrix with 2 rows and 1 columns
f = np.zeros((1,3)) # matrix with 1 row & 3 columns

g = e.dot(f)
print g.shape # outputs (2,3)
```

this ".T" "flips" the rows and columns of a matrix

```
h = np.zeros((5,4)).T # matrix with 4 rows and 5 columns
i = np.zeros((5,6)) # matrix with 6 rows & 5 columns

j = h.dot(i)
print j.shape # outputs (4,6)


h = np.zeros((5,4)) # matrix with 5 rows and 4 columns
i = np.zeros((5,6)) # matrix with 5 rows & 6 columns
j = h.dot(i)
print j.shape # throws an error
```

Conclusion

To predict, neural networks perform repeated weighted sums of the input.

We have seen an increasingly complex variety of neural networks in this chapter. I hope that it is clear that a relatively small number of simple rules are simply used repeatedly to create larger, more advanced neural networks. Furthermore, the intelligence of the network really depends on what weight values we give to our networks.

In the next chapter, we will be learning how to set our weights so that our neural networks make accurate predictions. We will find that in the same way that prediction is actually based on several simple techniques that are simply repeated/stacked on top of each other, "weight learning" is also a series of simple techniques that are simply combined many times across an architecture. See you there!

Introduction to Neural Learning
Gradient Descent

4

IN THIS CHAPTER

Do neural networks make accurate predictions?

Why measure error?

Hot and Cold Learning

Calculating both direction and amount from error

Gradient Descent

Learning is Just Reducing Error

Derivatives and how to use them to learn

Divergence and Alpha

> “The only relevant test of the validity of a hypothesis is comparison of prediction with experience.”
>
> — *MILTON FRIEDMAN*

Predict, Compare, and Learn

This chapter is about "Compare", and "Learn"

In Chapter 3, we learned about the paradigm: "Predict, Compare, Learn". In the previous chapter, we dove deep into the first part of this process "Predict". In this process we learned a myriad of things including the major parts of neural networks (nodes and weights), how datasets fit into networks (matching the number of datapoints coming in at one time), and finally how to use a neural network to make a prediction. Perhaps this process begged the question, "How do we set our weight values so that our network predicts accurately?". Answering this question will be the main focus of this chapter, covering the second two steps of our paradigm, "Compare", and "Learn".

Compare

A measurement of how much our prediction "missed".

Once we've made a prediction, the next step to learn is to evaluate how well we did. Perhaps this might seem like a rather simple concept, but we will eventually find that coming up with a good way to measure error is one of the most important and complicated subjects of Deep Learning.

In fact, there are many properties of "measuring error" that you have likely been doing your whole life without realizing it. Perhaps you (or someone you know) amplifies bigger errors while ignoring very small ones. In this chapter we will learn how to mathematically teach our network to do this. Furthermore (and this might seem too simple to be important), we will learn that error is always positive! We will consider the analogy of an "archer" hitting a target. Whether he is too low by and inch or too high by an inch, the error is still just 1 inch! In our neural network "Compare" step, we want to consider these kinds of properties when measuring error.

As a heads up, in this chapter we will only evaluate one, very simple way of measuring error called "Mean Squared Error". However, it is but one of many ways to evaluate the accuracy of your neural network.

As a closing thought, this step will give us a sense for "how much we missed", but this isn't enough to be able to learn. The output of our "compare" logic will simply be a "hot or cold" type signal. Given some prediction, we'll calculate an error measure that will either say "a lot" or "a little". It won't tell us why we missed, what direction we missed, or what we should do to fix it. It more or less just says "big miss", "little miss", or "perfect prediction". What we do about our error is captured in the next step, "Learn".

Learn

"Learning" takes our error and tells each weight how it can change to reduce it.

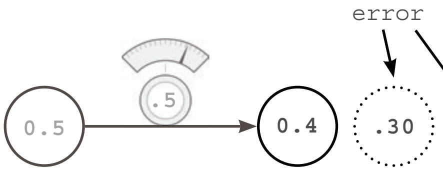
Learning is all about "error attribution", or the art of figuring out how each weight played its part in creating error. It's the "blame game" of Deep Learning. In this chapter, we will spend a great number of pages learning the most popular version of the Deep Learning "blame game" called Gradient Descent.

At the end of the day, it's going to result in computing a number for each of our weights. That number will represent how that weight should be higher or lower in order to reduce the error. Then we will move the weight according to that number, and we'll be done.

Compare: Does our network make good predictions?

Let's measure the error and find out!

Execute this code in your Jupyter notebook. It should print "0.3025".



```
knob_weight = 0.5
input = 0.5
goal_pred = 0.8

pred = input * knob_weight

error = (pred - goal_pred) ** 2

print(error)
```

The "error" is simply a way of measuring "how much we missed". There are multiple ways to calculate error as we will learn later. This one is "Mean Squared Error"

raw error

Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense.

What is the `goal_pred` variable?

Much like *input*, it's a number we recorded in the real world somewhere, but it's usually something that's hard to observe, like "the percentage of people who DID wear sweatsuits" given the temperature or "whether the batter DID in fact hit a home run" given his batting average.

Why is the error *squared*?

Think about an archer hitting a target. When he is 2 inches high, how much did he miss by? When he is two inches low, how much did he miss by? Both times he only missed by 2 inches. The primary reason why we *square* "how much we missed" is that it forces the output to be *positive*. `pred-goal_pred` could be negative in some situations... *unlike actual error.*

Doesn't squaring make big errors (>1) bigger and small errors (<1) smaller?

Yeah...It is kindof a weird way of measuring error... but it turns out that **amplifying** big errors and **reducing** small errors is actually ok. Later, we'll use this error to help the network learn... and we'd rather it *pay attention* to the big errors and not worry so much about the small ones. Good parents are like this too. They practically ignore errors if they're small enough (i.e. breaking the lead on your pencil) but might go nuclear for big errors (i.e. crashing the car). See why squaring is valuable?

Why measure error?

Measuring error simplifies the problem.
The goal of training our neural network is to make correct predictions. That's what we want. And in the most pragmatic world (as mentioned in the last chapter), we want the network to take input that we can easily calculate (today's stock price), and predict things that are hard to calculate (tomorrow's stock price). That's what makes a neural network useful.

It turns out that "changing knob_weight to make the network correctly predict the goal_prediction" is *slightly* more complicated than "changing the knob_weight to make error == 0" There's something more concise about looking at the problem this way. Ultimately, both of those statements say the same thing, but trying to *get the error to 0* just seems a bit more straightforward.

Different ways of measuring error *prioritize error differently*.
If this is a bit of a stretch right now, that's ok... but think back to what I said on the last page. By *squaring* the error, numbers that are less than 1 get *smaller* whereas numbers that are greater than 1 get *bigger*. This means that we're going to change what I call **"pure error" (prediction-goal_prediction)** so that bigger errors become VERY big and smaller errors quickly become irrelevant. By measuring error this way, we can *prioritize* big errors over smaller ones. When we have somewhat large "pure errors" (say... 10), we're going to tell ourselves we have *very* large error (10**2 == 100), and in contrast, when we have small "pure errors" (say... 0.01), we're going to tell ourselves that we have *very* small error (0.01 **2 == 0.0001). See what I mean about *prioritizing*? It's just modifying what we *consider to be error* so that we amplify big ones and largely ignore small ones. In contrast, if we took the *absolute value* instead of *squaring* the error, we wouldn't have this type of prioritization. The error would just be the positive version of the "pure error"... which would be fine... just different. More on this later.

Why do we only want *positive* error?
Eventually, we're going to be working with *millions* of input -> goal_prediction pairs... and we're still going to want to make accurate predictions. This means that we're going to try to take the *average error* down to 0.

This presents a problem if our error can be positive and negative. Imagine if we had two datapoints... two input -> goal_prediction pairs that we were trying to get the neural network to correctly predict. If the first had an error of 1,000, and the second had an error of -1,000, then our *average error* would be ZERO! We would fool ourselves into thinking we predicted perfectly when we missed by 1000 each time!!! This would be really bad. Thus, we want the error of *each prediction* to always be *positive* so that they don't accidentally cancel each other out when we average them.

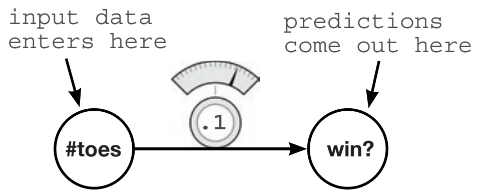
What's the Simplest Form of Neural Learning?

Learning using the Hot and Cold Method

At the end of the day, learning is really about one thing, adjusting our `knob_weight` either up or down so that our `error` reduces. If we keep doing this and our `error` goes to 0, we are done learning! So, how do we know whether to turn the knob up or down? Well, we try *both up and down* and see which one reduces the error! Whichever one reduces the error is used to actually update the `knob_weight`. It's simple, but effective. After we do this over and over again, eventually our error==0, which means our neural network is predicting with perfect accuracy.

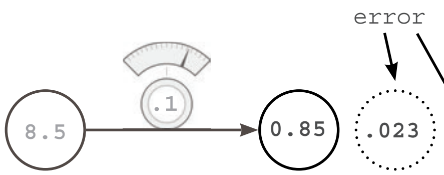
> **Hot and Cold Learning**
>
> Wiggling our weights to see which direction reduces the error the most, moving our weights in that direction, and repeating until the error gets to 0.

1 An Empty Network



```
weight = 0.1

lr = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

2 PREDICT: Making A Prediction And Evaluating Error



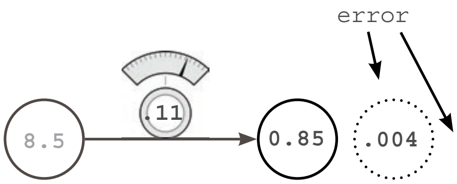
```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
true = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - true) ** 2
```

The "error" is simply a way of measuring "how much we missed". There are multiple ways to calculate error as we will learn later. This one is "Mean Squared Error"

raw error

Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense.

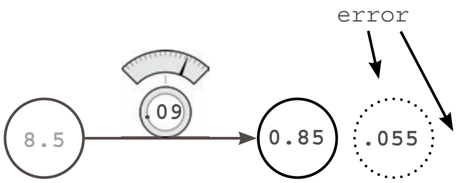
3 COMPARE: Making A Prediction With a Higher Weight And Evaluating Error



```
lr = 0.01

p_up = neural_network(input,weight+lr)

e_up = (p_up - true) ** 2
```

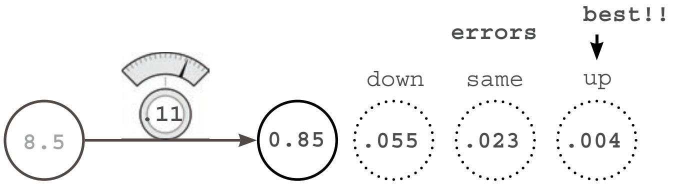
We want to move the weight so that the error goes downward, so we're going to try moving the weight up and down to see which one has the lowest error. First, we're trying moving the weight up (weight+lr).

4 COMPARE: Making A Prediction With a Lower Weight And Evaluating Error



```
lr = 0.01

p_dn = neural_network(input,weight-lr)

e_dn = (p_dn - true) ** 2
```

5 COMPARE + LEARN: Comparing our Errors and Setting our New Weight



```
if(error > e_dn ||
   error > e_up):

  if(e_dn < e_up):
     weight -= lr

  if(e_up < e_up):
     weight += lr
```

These last 5 steps comprise 1 iteration of Hot and Cold Learning. Fortunately, this iteration got us pretty close to the correct answer all by itself. (The new error is only 0.004). However, under normal circumstances, we would have to repeat this process many times in order to find the correct weights. Some people even have to train their networks for weeks or months before they find a good enough weight configuration.

This reveals what learning in neural networks really is. It's a **search problem**. We are *searching* for the best possible configuration of weights so that our network's error falls to zero (and predicts perfectly). As with all other forms of search, we might not find exactly what we're looking for, and even if we do, it may take some time. On the next page, we'll use Hot and Cold Learning for a *slightly* more difficult prediction so that you can see this searching in action!

Hot and Cold Learning

Perhaps the simplest form of learning.

Execute this code in your Jupyter Notebook. (New neural network modifications are in **bold**.) This code attempts to correctly predict 0.8.

```
weight = 0.5
input = 0.5
goal_prediction = 0.8

step_amount = 0.001

for iteration in range(1101):

    prediction = input * weight
    error = (prediction - goal_prediction) ** 2

    print "Error:" + str(error) + " Prediction:" + str(prediction)

    up_prediction = input * (weight + step_amount)
    up_error = (goal_prediction - up_prediction) ** 2

    down_prediction = input * (weight - step_amount)
    down_error = (goal_prediction - down_prediction) ** 2

    if(down_error < up_error):
        weight = weight - step_amount

    if(down_error > up_error):
        weight = weight + step_amount
```

how much to move our weights each iteration

repeat learning many times so that our error can keep getting smaller

TRY UP!

TRY DOWN!

If down is better, go down!

If up is better, go up!

When I run this code, I see the following output:

```
Error:0.3025 Prediction:0.25
Error:0.30195025 Prediction:0.2505
            ....
Error:2.50000000033e-07 Prediction:0.7995
Error:1.07995057925e-27 Prediction:0.8
```

Our last step correctly predicts 0.8!

Characteristics of Hot and Cold Learning

It's simple

Hot and Cold learning is simple. After making our prediction, we predict two more times, once with a slightly higher weight and again with a slightly lower weight. We then move the `weight` depending on which **direction** gave us a smaller `error`. Repeating this enough times eventually reduces our `error` down to 0.

> **Why did I iterate exactly 1101 times?**
> The neural network reaches 0.8 after exactly that many iterations. If you go past that, it wiggles back and forth between 0.8 and just above/below 0.8... making for a less pretty error log printed at the bottom of the left page. Feel free to try it out though.

PROBLEM #1: It's inefficient

We have to predict *multiple times* in order to make a single *knob_weight* update. This seems very inefficient.

PROBLEM #2: Sometimes it's impossible to predict the *exact* goal prediction.

With a set `step_amount`, unless the perfect `weight` is exactly `n*step_amount` away, the network will eventually overshoot by some number less than `step_amount`. When it does so, it will then start alternating back and forth between each side of the `goal_prediction`. Set the `step_amount` to 0.2 to see this in action. If you set `step_amount` to 10 you'll really break it! When I try this I see the following output. It never *remotely* comes *close* to 0.8!!!

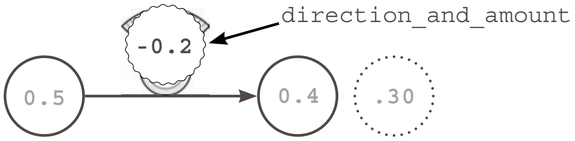
The real problem here is that even though we know the correct **direction** to move our `weight`, we *don't know the correct **amount***. Since we don't *know* the correct amount, we just pick a fixed one at random (`step_amount`). Furthermore, this *amount* has NOTHING to do with our `error`. Whether our `error` is BIG or our `error` is TINY, our `step_amount` is the same. So, Hot and Cold Learning is kindof a bummer... it's inefficient because we *predict 3 times for each weight update* and our *step_amount* is completely arbitrary... which can prevent us from learning the correct `weight` value.

```
Error:0.3025 Prediction:0.25
Error:19.8025 Prediction:5.25
Error:0.3025 Prediction:0.25
Error:19.8025 Prediction:5.25
Error:0.3025 Prediction:0.25
....
.... repeating infinitely...
```

What if we had a way of computing both **direction** and **amount** for each `weight` without having to repeatedly make predictions?

Calculating Both *direction* and *amount* from *error*

Let's measure the error and find out!
Execute this code in your Jupyter notebook.



(1) "pure error"

(2) scaling, negative reversal, and stopping

```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount

    print "Error:" + str(error) + " Prediction:" + str(pred)
```

What you see above is a *superior* form of learning known as **Gradient Descent**. This method allows us to (in a single line of code... seen above in **bold**) calculate both the *direction* and the *amount* that we should change our `weight` so that we reduce our `error`.

What is the `direction_and_amount`?
It represents how we want to change our `weight`. The first **(1)** is what we call "pure error" which equals (**`pred - goal_pred`**). This number represents "the raw direction and amount that we missed". The second part **(2)** is the multiplication by the **`input`** which performs scaling, negative reversal and stopping...modifying the "pure error" so that it's ready to update our `weight`.

What is the "pure error"?
It's the (**`pred`** - **`goal_pred`**) which indicates "the raw direction and amount that we missed". If this is a *positive* number then we predicted too *high* and vice versa. If this is a *big* number then we missed by a *big* amount, etc.

What is "scaling, negative reversal, and stopping"?
These three attributes have the combined affect of translating our "pure error" into "the absolute amount that we want to change our `weight`". They do so by addressing three *major edge cases* at which points the "pure error" is not sufficient to make a good modification to our `weight`.

What is "stopping"?
This is the first (and simplest) affect on our "pure error" caused by multiplying it by our `input`. Imagine plugging in a CD player into your stereo. If you turned the volume all the way up but the CD player was *off*... it simply wouldn't matter. "Stopping" addresses this in our neural network... if our `input` is 0, then it will force our direction_and_amount to also be 0. We don't learn (i.e. "change the volume") when our `input` is 0 because there's nothing to learn... every `weight` value has the same `error`... and moving it makes no difference because the `pred` is always 0.

What is "negative reversal"?
This is probably our most difficult and important effect. Normally (when `input` is positive), moving our `weight` *upward* makes our prediction move *upward.* However, if our `input` is *negative*, then all of a sudden our `weight` changes directions!!! When our `input` is *negative*, then moving our `weight` *up* makes the prediction go *down*. It's reversed!!! How do we address this? Well, multiplying our "pure error" by our *input* will *reverse the sign* of our `direction_and_amount` in the event that our `input` is negative. This is "negative reversal", ensuring that our `weight` moves in the correct direction, even if the `input` is negative.

What is "scaling"?
Scaling is the second effect on our "pure error" caused by multiplying it by our **`input`**. Logically, it means that if our input was big, our weight update should also be big. This is more of a "side affect" as it can often go out of control. Later, we will use *alpha* to address when this scaling goes out of control.

When you run the code in the top left, you should see the following output.

```
Error:0.3025 Prediction:0.25
Error:0.17015625 Prediction:0.3875
Error:0.095712890625 Prediction:0.490625
                   ...

Error:1.7092608064e-05 Prediction:0.79586567925
Error:9.61459203602e-06 Prediction:0.796899259437
Error:5.40820802026e-06 Prediction:0.797674444578
```

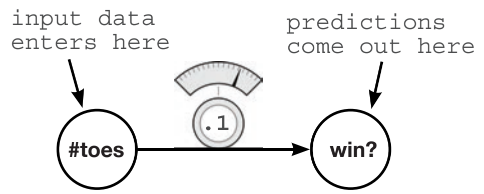
Our last steps correctly approach 0.8!

In this example, we saw Gradient Descent in action in a bit of an oversimplified environment. On the next page, we're going to see it in it's more native environment. Some terminology will be different, but we will code it in a way that makes it more obviously applicable to other kinds of networks (such as those with multiple inputs and outputs)

One Iteration of Gradient Descent

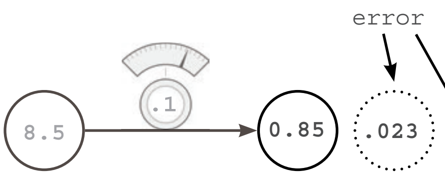
This performs a weight update on a single "training example" (input->true) pair

1 An Empty Network



```
weight = 0.1

alpha = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

2 PREDICT: Making A Prediction And Evaluating Error



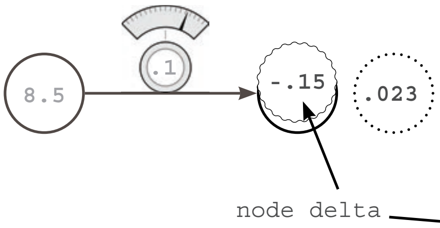
```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2
```

raw error

Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense.

The "error" is simply a way of measuring "how much we missed". There are multiple ways to calculate error as we will learn later. This one is "Mean Squared Error"

3 COMPARE: Calculating "Node Delta" and Putting it on the Output Node

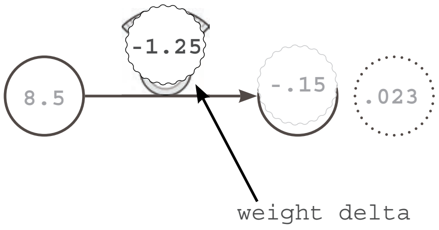


```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred
```

Delta is a measurement of "how much this node missed". Thus, since the true prediction was 1.0, and our network's prediction was 0.85, the network was too *low* by 0.15. Thus, delta is ***negative*** 0.15.

The primary difference between the gradient descent on the previous page and the implementation on this page just happened. `delta` is a new variable. It's the "raw amount that the node was too high or too low". Instead of computing `direction_and_amount` directly, we first calculate how much we wanted our output node to be different. Only then do we compute our `direction_and_amount` to change the `weight` (in step 4, now renamed "`weight_delta`").

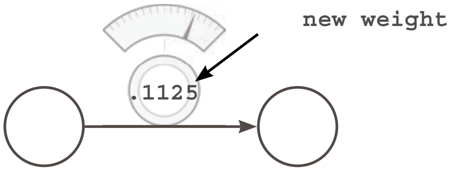
4 LEARN: Calculating "Weight Delta" and Putting it on the Weight



```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = input * delta
```

Weight delta is a measure of "how much this weight caused the newtork to miss". We calculate it by multiplying the weight's output "Node Delta" by the weight's input. Thus, we create each "Weight Delta" by *scaling* it's output "Node Delta" by the weight's input. This accounts for the 3 aforementioned properties of our "direction_and_amount", scaling, negative reversal, and stopping.

5 LEARN: Updating the Weight



```
number_of_toes = [8.5]
win_or_lose_binary = [1] // (won!!!)


input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)


error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = input * delta


alpha = 0.01 // fixed before training

weight -= weight_delta * alpha
```

We multiply our weight_delta by a small number "alpha" before using it to update our weight. This allows us to control how fast the network learns. If it learns too fast, it can update weights too aggressively and overshoot. More on this later. Note that the weight update made the same change (small increase) as Hot and Cold Learning

Learning Is Just Reducing Error

Modifying weight to reduce our error.

Putting together our code from the previous pages. We now have the following:

```
weight, goal_pred, input = (0.0, 0.8, 0.5)

for iteration in range(4):

    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta
    print "Error:" + str(error) + " Prediction:" + str(pred)
```

these lines have a *secret*

The Golden Method for Learning

Adjusting each `weight` in the correct *direction* and by the correct *amount* so that our `error` reduces to 0.

All we're trying to do is figure out the right **direction** and **amount** to modify `weight` so that our `error` goes down. The secret to this lies in our `pred` and `error` calculations. Notice that we actually use our `pred` *inside* the `error` calculation. Let's replace our `pred` variable with the code we used to generate it.

```
error = ((input * weight) - goal_pred) ** 2
```

This doesn't change the value of error at all! It just combines our two lines of code so that we compute our error directly. Now, remember that our input and our goal_prediction are actually fixed at 0.5 and 0.8 respectively (we set them before the network even starts training). So, if we replace their variables names with the values... the *secret* becomes clear

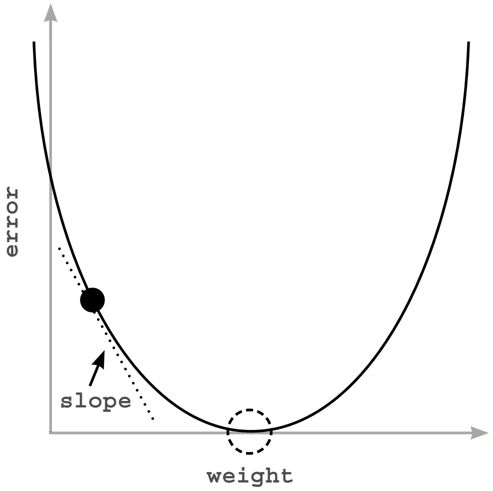
```
error = ((0.5 * weight) - 0.8) ** 2
```

The Secret

For any `input` and `goal_pred`, there is an *exact relationship* defined between our **`error`** and **`weight`**, found by combining our **`prediction`** and **`error`** formulas. In this case:

```
error = ((0.5 * weight) - 0.8) ** 2
```

Let's say that you moved `weight` up by 0.5... if there is an *exact relationship* between `error` and `weight`... we should be able to calculate how much this also *moves* the `error`! What if we wanted to *move* the `error` in a specific direction? Could it be done?



This graph represents *every value of error* for *every weight* according to the relationship in the formula above. Notice it makes a nice *bowl shape*. The black "dot" is at the point of BOTH our current `weight` and `error`. The dotted "circle" is where we want to be (`error` == 0).

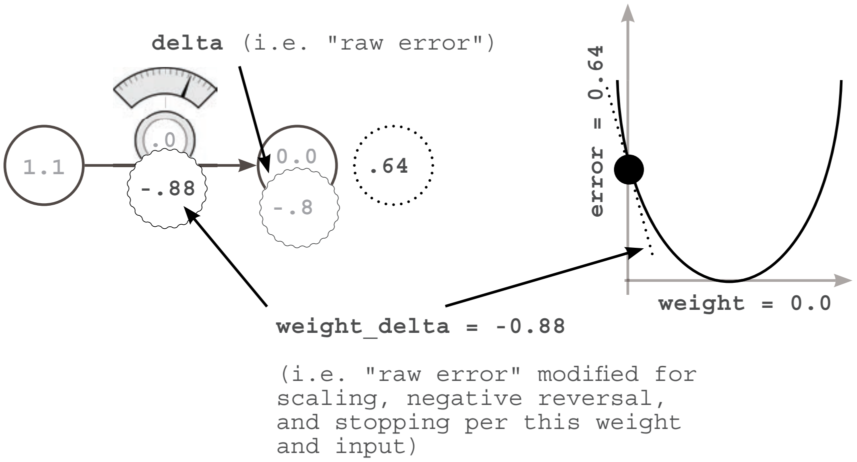
Key Takeaway: The *slope* points to the bottom of the bowl (lowest `error`) *no matter where you are in the bowl*. We can use this *slope* to help our neural network *reduce the error*.

Let's Watch Several Steps of Learning

Will we eventually find the bottom of the bowl?

```
weight, goal_pred, input = (0.0, 0.8, 1.1)

for iteration in range(4):
    print "-----\nWeight:" + str(weight)
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta
    print "Error:" + str(error) + " Prediction:" + str(pred)
    print "Delta:" + str(delta) + " Weight Delta:" + str(weight_delta)
```

1 **A Big Weight Increase**

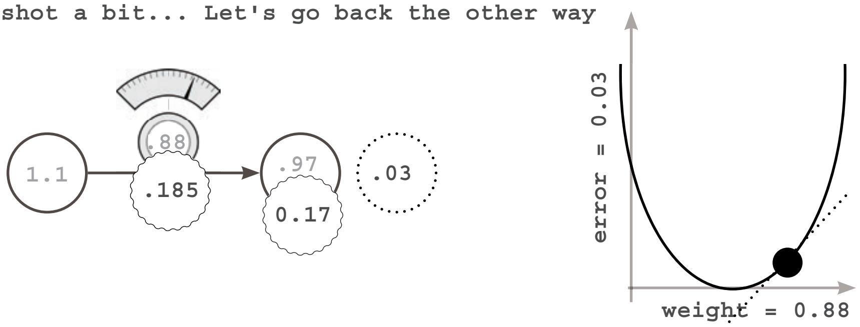


delta (i.e. "raw error")

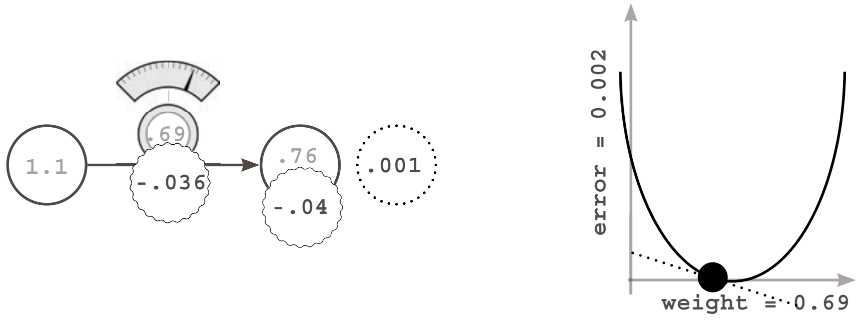
weight_delta = -0.88

(i.e. "raw error" modified for scaling, negative reversal, and stopping per this weight and input)

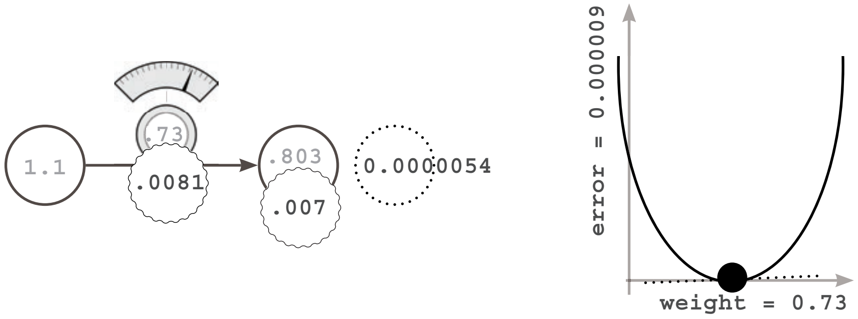
2 **Overshot a bit... Let's go back the other way**



3 Overshot Again! Let's go back again... but only just a little



4 Ok, we're pretty much there...



Code Output

```
-----
Weight:0.0
Error:0.64 Prediction:0.0
Delta:-0.8 Weight Delta:-0.88
-----
Weight:0.88
Error:0.028224 Prediction:0.968
Delta:0.168 Weight Delta:0.1848
-----
Weight:0.6952
Error:0.0012446784 Prediction:0.76472
Delta:-0.03528 Weight Delta:-0.038808
-----
Weight:0.734008
Error:5.489031744e-05 Prediction:0.8074088
Delta:0.0074088 Weight Delta:0.00814968
```

Why does this work? What really is weight_delta?

Let's back up and talk about functions. What is a function? How do we understand it?

Consider this function:

```
def my_function(x):
    return x * 2
```

A function takes some numbers as input and gives you another number as output. As you can imagine, this means that the function actually defines some sort of *relationship* between the input number(s) and the output number(s). Perhaps you can also see why the ability to *learn a function* is so powerful... it allows us to take some numbers (say...image pixels) and convert them into other numbers (say... the *probability* that the image contains a cat).

Now, every function has what you might call *moving parts*. It has pieces that we can tweak or change to make the ouput that the function generates *different*. Consider our "my_function" above. Ask yourself, "what is controlling the relationship between the input and the output of this function?". Well, it's the 2! Ask the same question about the function below.

```
error = ((input * weight) - goal_pred) ** 2
```

What is controlling the relationship between the `input` and the output (`error`)? Well, plenty of things are! This function is a bit more complicated! `goal_pred`, `input`, `**2`, `weight`, and all the parenthesis and algebraic operations (addition, subtraction, etc.) play a part in calculating the error... and tweaking any one of them would *change* the error. This is important to consider.

Just as a thought exercise, consider changing your `goal_pred` to reduce your error. Well, this is silly... but totally doable! In life, we might call this "giving up"... setting your goals to be whatever your capability is. It's just denying that we missed! This simply wouldn't do.

What if we changed the `input` until our error went to zero... well... this is akin to seeing the world as you *want* to see it instead of as it actualy is. This is changing your *input data* until you're predicting what you want to predict (sidenote: this is loosely how "inceptionism works").

Now consider changing the 2... or the additions...subtractions... or multiplications... well this is just changing how you calculate error in the first place! Our error calculation is meaningless if it doesn't actually give us a good measure of *how much we missed* (with the right properties mentioned a few pages ago). This simply won't do either.

So, what do we have left? The only variable we have left is our `weight`. Adjusting this doesn't change our perception of the world... doesn't change our goal... and doesn't destroy our error measure. In fact, changing weight means that the function *conforms to the patterns in the data*. By forcing the rest of our function to be *unchanging*, we force our function to correctly model some pattern in our data. It is only allowed to modify how the network *predicts*.

So, at the end of the day, we're modifying specific parts of an error function until the `error` value goes to zero. This error function is calculated using a combination of variables... some of them we can change (weights) and some of them we cannot (input data, output data, and the error logic itself).

```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount

    print "Error:" + str(error) + " Prediction:" + str(pred)
```

We can modify *anything* in our `pred` calculation except the `input`.

In fact, we're going to spend *the rest of this book* and many deep learning researchers will spend *the rest of their lives* just trying everything you can imagine to that `pred` calculation so that it can make good predictions. Learning is all about automatically changing that prediction function so that it makes good predictions... aka... so that the subsequent `error` goes down to 0.

Ok, now that we know what we're *allowed* to change... how do we actually go about doing that changing? That's the good stuff! That's the *machine learning*, right? In the next, section, we're going to talk about exactly that.

Tunnel Vision on One Concept

Concept: "Learning is adjusting our weight to reduce the error to zero"

So far in this chapter, we've been hammering on the idea that learning is really just about adjusting our `weight` to reduce our error to zero. This is the secret sauce. Truth be told, knowing how to do this is *all about* understanding the **relationship** between our `weight` and our `error`. If we understand this relationship, we can know how to adjust our `weight` to reduce our `error`.

What do I mean by "understand the relationship"? Well, to understand the relationship between two variables is really just to understand *how changing one variable changes the other*. In our case, what we're really after is the **sensitivity** between these two variables. Sensitivity is really just another name for *direction* and *amount*. We want to know how sensitive the `error` is to the `weight`. We want to know the *direction* and the *amount* that the `error` changes when we change the `weight`. This is the goal. So far, we've used two different methods to attempt to understand this relationship.

You see, when we were "wiggling" our `weight` (hot and cold learning) and studying its affect on our `error`, we were really just *experimentally* studying the relationship between these two variables. It's like when you walk into a room with 15 different unlabeled light switches. You just start flipping them on and off to learn about their relationship to various lights in the room. We did the same thing to study the relationship between our `weight` and our `error`. We just wiggled the `weight` up and down and watched for how it changed the `error`. Once we knew the relationship, we could move the `weight` in the right direction using two simple if statements.

```
if(down_error < up_error):
    weight = weight - step_amount

if(down_error > up_error):
    weight = weight + step_amount
```

Now, let's go back to the formula from the previous pages, where we combined our pred and error logic. As mentioned, they quietly define an *exact relationship* between our `error` and our `weight`.

```
error = ((input * weight) - goal_pred) ** 2
```

This line of code, ladies and gentlemen, is the secret. This is a formula. This is the relationship between `error` and `weight`.This relationship is exact. It's computable. It's universal. It is and it will always be. Now, how can we use this formula to know how to change our `weight` so that our `error` moves in a *particular direction*. Now THAT is the right question! Stop. I beg you. Stop and appreciate this moment. This formula is the exact relationship between these two variables, and now we're going to figure out how to change one variable so that we move the other variable in a particular direction. As it turns out, there's a method for doing this for *any* formula. We're going to use it for reducing our error.

A Box With Rods Poking Out of It

An analogy.

Picture yourself sitting in front of a cardboard box that has two circular rods sticking through two little holes. The blue rod is sticking out of the box by 2 inches, and the red rod is sticking out of the box by 4 inches. Imagine that I told you that these rods were connected in some way, but I wouldn't tell you in what way. You had to experiment to figure it out.

So, you take the blue rod and push it in 1 inch, and watch as... while you're pushing... the red rod also moves into the box by 2 inches!!! Then, you pull the blue rod back out an inch, and the red rod follows again!!... pulling out by 2 inches. What did you learn? Well, there seems to be a *relationship* between the red and blue rods. However much you move the blue rod, the red rod will move by twice as much. You might say the following is true.

```
red_length = blue_length * 2
```

As it turns out, there's a formal definition for "when I tug on this part, how much does this other part move". It's called a **derivative** and all it really means is "how much does rod X move when I tug on rod Y."

In the case of the rods above, the derivative for "how much does red move when I tug on blue" is 2. Just 2. Why is it 2? Well, that's the *multiplicative* relationship determined by the formula.

```
                                  derivative
red_length = blue_length * 2
```

Notice that we always have the derivative *between two variables*. We're always looking to know how one variable moves when we change another one! If the derivative is *positive* then when we change one variable, the other will move in the *same* direction! If the derivative is *negative* then when we change one variable, the other will move in the *opposite* direction.

Consider a few examples. Since the derivative of red_length compared to blue_length is 2, then both numbers move in the same direction! More specifically, red will move *twice as much* as blue in the same direction. If the derivative had been -1, then red would move in the *opposite* direction by the same amount. Thus, given a function, the derivative represents the **direction** and the **amount** that one variable changes if you change the other variable. This is exactly what we were looking for!

Derivatives... take Two

Still a little unsure about them?... let's take another perspective...

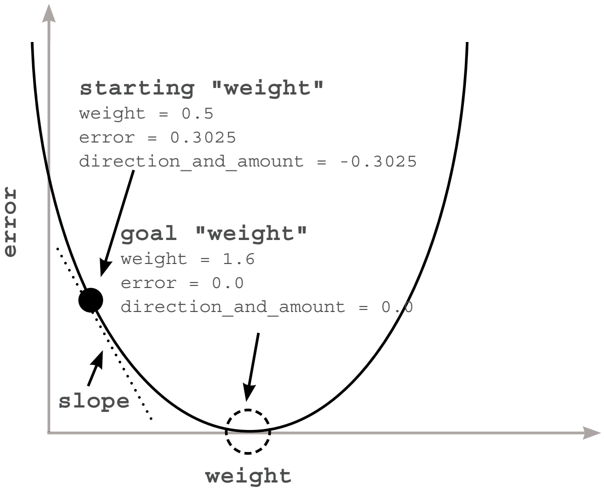
There are two ways I've heard people explain derivatives. One way is all about understanding "how one variable in a function changes when you move another variable". The other way of explaining it is "a derivative is the slope at a point on a line or curve". As it turns out, if you take a function and plot it out (draw it), the slope of the line you plot is the *same thing* as "how much one variable changes when you change the other". Let me show you by plotting our favorite function.

```
error = ((input * weight) - goal_pred) ** 2
```

Now remember... our `goal_pred` and `input` are fixed, so we can rewrite this function:

```
error = ((0.5 * weight) - 0.8) ** 2
```

Since there are only two variables left that actually change (all the rest of them are fixed), we can just take every `weight` and compute the `error` that goes with it! Let's plot them As you can see on the right, our plot looks like a big U shaped curve! Notice that there is also a point in the middle where the `error` == 0! Also notice that to the right of that point, the slope of the line is *positive*, and to the left of that point, the slope of the line is *negative*. Perhaps even more interesting, the farther away from the **goal weight** that you move, the *steeper* the slope gets. We like all of these properties. The *slope*'s *sign* gives us **direction** and the slope's *steepness* gives us **amount**. We can use both of these to help find the goal `weight`.



Even now, when I look at that curve, it's easy for me to lose track of what it represents. It's actually similar to our "hot and cold" method for learning. If we just tried *every possible value* for `weight`, and plotted it out, we'd get this curve. And what's really remarkable about derivatives is that they can see past our big formula for computing `error` (at the top of this page) and see this curve! We can actually compute the **slope** (i.e. derivative) of the line for any value of `weight`. We can then use this slope (derivative) to figure out which **direction** reduces our `error`! Even better, based on the *steepness* we can get at least some idea for how far away we are (although not an exact one... as we'll learn more about later).

What you really need to know...

With derivatives... we can pick any two variables... in any formula... and know how they interact.

Take a look at this *big whopper of a function.*

```
y = (((beta * gamma) ** 2) + (epsilon + 22 - x)) ** (1/2)
```

Here's what you need to know about derivatives. For any function (even this whopper) you can pick any two variables and understand their relationship with each other. For any function, you can pick two variables and plot them on an x-y graph like we did on the last page. For any function, you can pick two variables and compute how much one changes when you change the other. Thus, for any function, we can learn how to change one variable so that we can move another variable in a direction. Sorry to harp on, but it's important you know this in your bones.

Bottom Line: In this book we're going to build neural networks. A neural network is really just one thing... a bunch of **weights** which we use to compute an **error** function. And for any error function (no matter how complicated), we can compute the relationship between any `weight` and the final `error` of the network. With this information, we can change each `weight` in our neural network to reduce our `error` down to 0... and that's exactly what we're going to do.

What you don't really need to know...

....Calculus....

So, it turns out that learning all of the methods for taking any two variables in any function and computing their relationship takes about 3 semesters of college. Truth be told, if you went through all three semesters so that you could learn how to do Deep Learning... you'd only actually find yourself *using* a very small subset of what you learned. And really, Calculus is just about memorizing and practicing every possible derivative rule for every possible function.

So, in this book I'm going to do what I typically do in real life (cuz i'm lazy?... i mean... efficient?) ... just look up the derivative in a reference table. All you really *need to know* is what the derivative *represents*. It's the relationship between two variables in a function so that you can know how much one changes when you change the other. It's just the sensitivity between two variables. I know that was a lot of talking to just say "It's the sensitivity between two variables"... but it is. Note that this can include both "positive" sensitivity (when variables move together) and "negative" sensitivity (when they move in opposite directions) or "zero" sensitivity...where one stays fixed regardless of what you do to the other. For example, y = 0 * x. Move x... y is always 0. Ok, enough about derivatives. Let's get back to Gradient Descent.

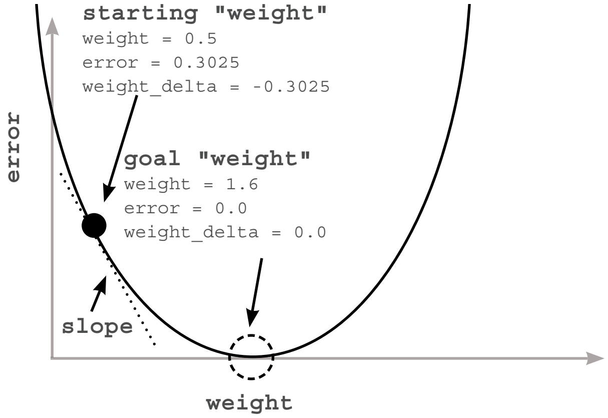
How to use a derivative to learn

"weight_delta" is our derivative.

What is the difference between the `error` and the derivative of our `error` and `weight`? Well the error is just a *measure* of how much we missed. The derivative defines the *realtionship* between each weight and how much we missed. In other words, it tells *how much changing a weight contributed to the error*. So, now that we know this, how do we use it to move the error in a particular direction?

So, we've learned the relationship between two variables in a function... how do we exploit that relationship? As it turns out, this is incredibly visual and intuitive. Check out our error curve again. The black dot is where our `weight` starts out at (0.5). The dotted circle is where we want it to go... our goal `weight`. Do you see the dotted line attached to our black dot? That's our *slope* otherwise known as our *derivative*. It tells us *at that point in the curve* how much the `error` changes when we change the `weight`. Notice that it's pointed downward! It's a negative slope!

The slope of a line or curve *always points* in the opposite direction to the *lowest point* of the line or curve. So, if you have a negative slope, you *increase* your `weight` to find the minimum of the `error`. Check it out!



So, how do we use our *derivative* to find the `error` minimum (lowest point in the `error` graph)? We just move the opposite direction of the slope! We move in the opposite direction of the derivative! So, we can take each weight, calculate the derivative of that weight with respect to the error (so we're comparing two variables there... the weight and the error) and then change the weight in the *opposite* direction of that slope! That will move us to the minimum!

Let's remember back to our goal again. We are trying to figure out the **direction** and the **amount** to change our `weight` so that our `error` goes down. A derivative gives us the relationship between any two variables in a function. We use the derivative to determine the relationship between any *weight* and the *error*. We then move our `weight` in the *opposite* direction of the derivative to find the lowest `weight`. Wallah! Our neural network learns!

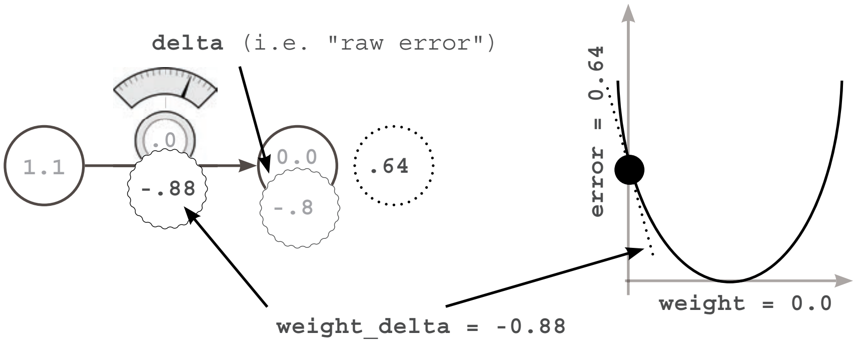
This method for learning (finding error minimums) is called **Gradient Descent**. This name should seem intuitive! We move in the `weight` value *opposite the gradient* value, which *descends* our error to 0. By *opposite,* I simply mean that we increase our `weight` when we have a negative gradient and vice versa. It's like gravity!

Look Familiar?

```
weight = 0.0
goal_pred = 0.8
input = 1.1

for iteration in range(4):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta

    print "Error:" + str(error) + " Prediction:" + str(pred)
```

derivative
(i.e., how fast the error changes given changes in the weight)

① **A Big Weight Increase**

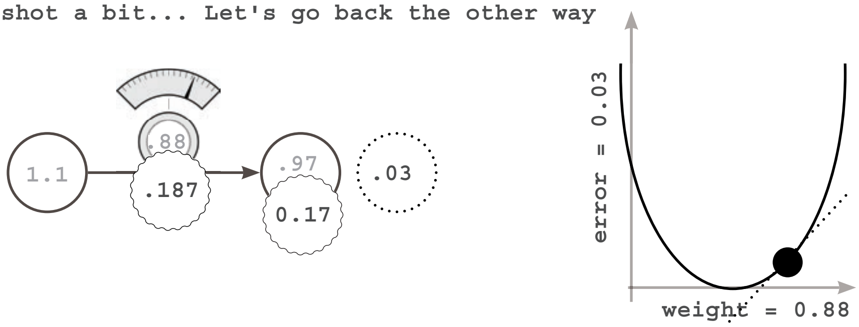


delta (i.e. "raw error")

weight_delta = -0.88

(i.e. "raw error" modified for scaling, negative reversal, and stopping per this weight and input)

② **Overshot a bit... Let's go back the other way**



Breaking Gradient Descent

Just Give Me The Code

```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = input * delta
    weight = weight - weight_delta
    print "Error:" + str(error) + " Prediction:" + str(pred)
```

When I run this code, I see the following output...

```
Error:0.3025 Prediction:0.25
Error:0.17015625 Prediction:0.3875
Error:0.095712890625 Prediction:0.490625
                    ...

Error:1.7092608064e-05 Prediction:0.79586567925
Error:9.61459203602e-06 Prediction:0.796899259437
Error:5.40820802026e-06 Prediction:0.797674444578
```

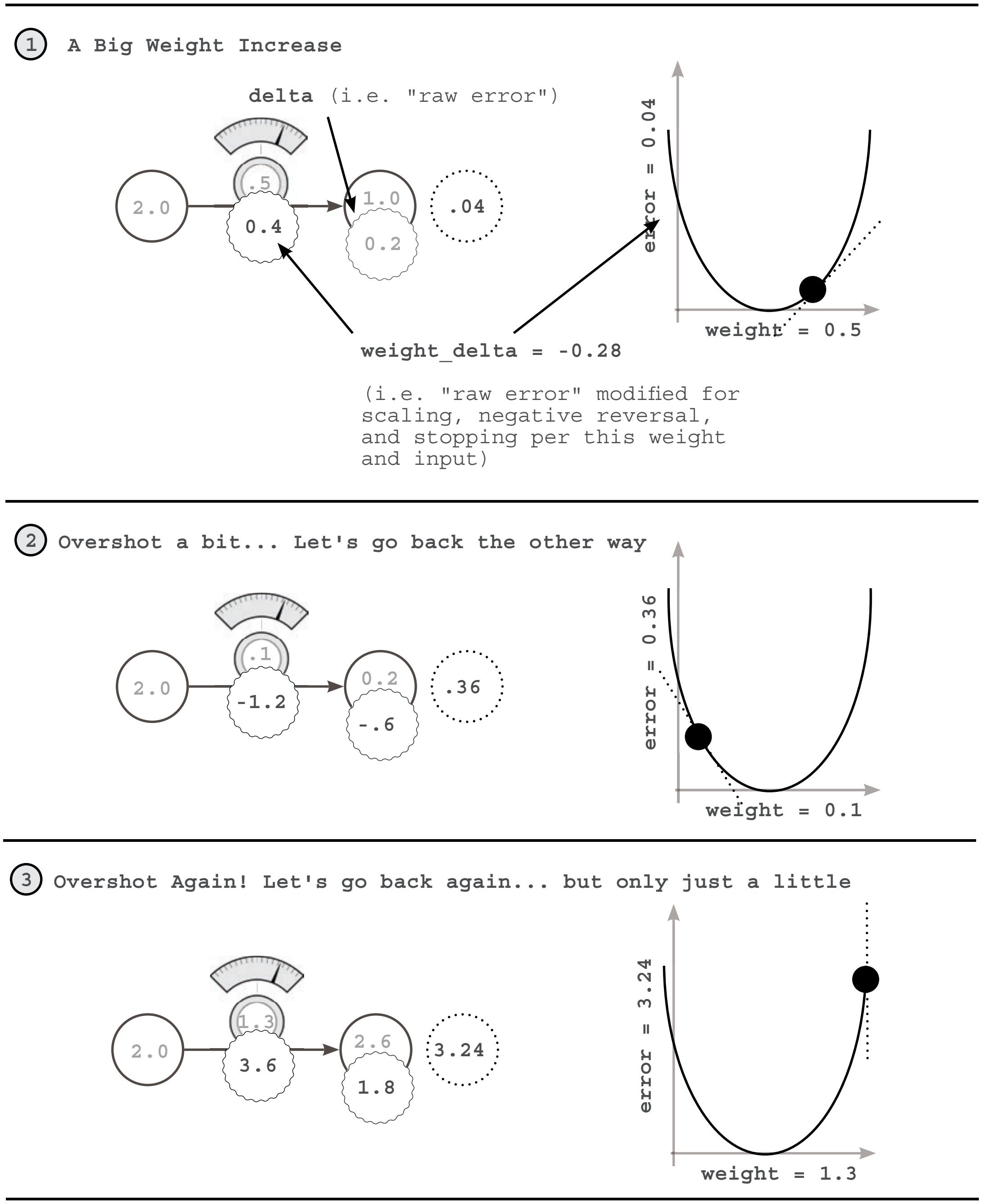
Now that it works... let's break it! Play around with the starting `weight`, `goal_pred`, and `input` numbers. You can set them all to just about anything and the neural network will figure out how to predict the output given the input using the weight. See if you can find some combinations that the neural network cannot predict! I find that trying to break something is a great way to learn about it.

Let's try setting `input` to be equal to 2, but still try to get the algorithm to predict 0.8. What happens? Well, take a look at the output.

```
Error:0.04 Prediction:1.0
Error:0.36 Prediction:0.2
Error:3.24 Prediction:2.6

           ...

Error:6.67087267987e+14 Prediction:-25828031.8
Error:6.00378541188e+15 Prediction:77484098.6
Error:5.40340687069e+16 Prediction:-232452292.6
```

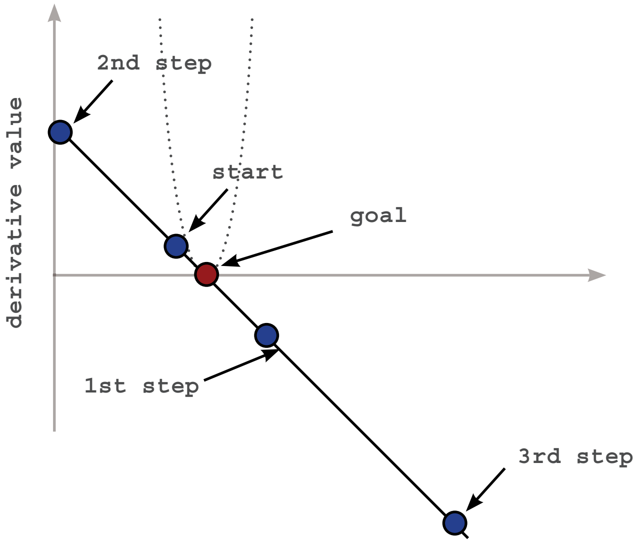
Woah! That's not what we wanted! Our predictions exploded! They alternate from negative to positive and negative to positive, getting farther away from the true answer at every step! In other words, every update to our weight **overcorrects**! In the next section, we'll learn more about how to combat this phenomenon.

Visualizing the Overcorrections


1
A Big Weight Increase
delta (i.e. "raw error")
2.0
.5
0.4
1.0
0.2
.04
error = 0.04
weight = 0.5
weight_delta = -0.28
(i.e. "raw error" modified for scaling, negative reversal, and stopping per this weight and input)
2
Overshot a bit... Let's go back the other way
2.0
.1
-1.2
0.2
-.6
.36
error = 0.36
weight = 0.1
3
Overshot Again! Let's go back again... but only just a little
2.0
1.3
3.6
2.6
1.8
3.24
error = 3.24
weight = 1.3

Divergence

Sometimes... neural networks explode in value... oops?



So what really happened? The explosion in error on the previous page is caused by the fact that we made the input larger. Consider how we're updating our weight.

```
weight = weight - (input * (pred - goal_pred))
```

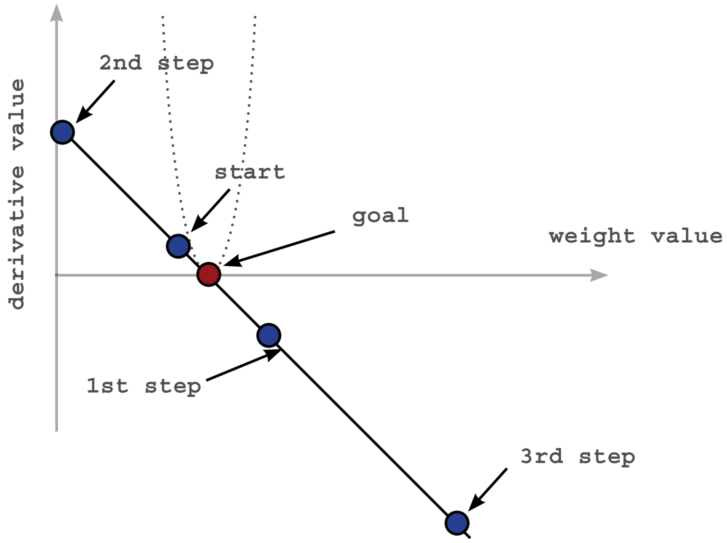
If our input is sufficiently large, this can make our weight update large even when our error is small. What happens when you have a large weight update and a small error? It overcorrects!!! If the new error is even bigger, it overcorrects even more!!! This causes the phenomenon that we saw on the previous page, called **divergence**.

You see, if we have a BIG input, then the prediction is very sensitive to changes in the weight (since `pred` = `input` * `weight`). This can cause our network to overcorrect. In other words, even though our `weight` is still only starting at 0.5, our derivative at that point is *very steep*. See how tight the u shaped error curve is in the graph above?

This is actually really intuitive. How do we predict? Well, we predict by *multiplying* our input by our `weight`. So, if our input is *huge*, then small changes in our `weight` are going to cause BIG changes in our prediction!! The error is very *sensitive* to our `weight`. Aka... the derivative is really big! So, how do we make it smaller?

Introducing.... Alpha

The simplest way to prevent overcorrecting our weight updates.



So, what was the problem we're trying to solve? The problem is this: if the input is too big, then our weight update can overcorrect. What is the symptom? The symptom is that when we overcorrect, our new derivative is even *larger in **magnitude*** than when we started (although the sign will be the opposite). Stop and consider this for a second. Look at the graph above to understand the symptom. The 2nd step is even farther away from the goal... which means the *derivative* is even greater in magnitude! This causes the 3rd step to be even farther away from the goal than the second step, and the neural network continues like this, demonstrating **divergence**.

The symptom is this overshooting. The solution is to *multiply the weight update by a fraction* to make it smaller. In most cases, this involves multiplying our weight update by a single real-valued number between 0 and 1, known as **alpha**. One might note, this has no affect on the *core issue* which is that our input is larger. It will also reduce the `weight` updates for inputs that aren't too large. In fact, finding the appropriate alpha, even for state-of-the-art neural networks, is often done simply by guessing. You watch your `error` over time. If it starts diverging (going up), then your alpha is too high, and you decrease it. If learning is happening too slowly, then your alpha is too low, and you increase it. There are other methods than simple gradient descent that attempt to counter for this, but gradient descent is still very popular.

Alpha In Code

Where does our "alpha" parameter come in to play?

So we just learned that **alpha** reduces our weight update so that it doesn't overshoot. How does this affect our code? Well, we were updating our weights according to the following formula.

```
weight = weight - derivative
```

Accounting for alpha is a rather small change, pictured below. Notice that if alpha is small (say...0.01), it will reduce our weight update considerably, thus preventing it from overshooting.

```
weight = weight - (alpha * derivative)
```

Well, that was easy! So, let's install alpha into our tiny implementation from the beginning of this chapter and run it where input = 2 (which previously didn't work)

```
weight = 0.5
goal_pred = 0.8
input = 2
alpha = 0.1

for iteration in range(20):
    pred  = input * weight
    error = (pred - goal_pred) ** 2
    derivative = input * (pred - goal_pred)
    weight = weight - (alpha * derivative)

    print "Error:" + str(error) + " Prediction:" + str(pred)
```

```
Error:0.04 Prediction:1.0
Error:0.0144 Prediction:0.92
Error:0.005184 Prediction:0.872

            ...

Error:1.14604719983e-09 Prediction:0.800033853319
Error:4.12576991939e-10 Prediction:0.800020311991
Error:1.48527717099e-10 Prediction:0.800012187195
```

What happens
when you make
alpha crazy
small or big?
What about
making it
negative?

Wallah! Our tiniest neural network can now make good predictions again! How did I know to set alpha to 0.1? Well, to be honest, I just tried it and it worked. And despite all the crazy advancements of deep learning in the past few years, most people just try several orders of magnitude of alpha (10,1,0.1,0.01,0.001,0.0001) and then tweak from there to see what works best. It's more art than science. There are more advanced ways which we can get to later, but for now, just try various alphas until you get one that seems to work pretty well. Play with it!

Memorizing

Ok... it's time to really learn this stuff

This may sound like something that's a bit intense, but I can't stress enough the value I have found from this exercise. The code on the previous page, see if you can build it in an iPython notebook (or a .py file if you must) from *memory*. I know that might seem like overkill, but I (personally) didn't have my *click* moment with neural networks until I was able to perform this task.

Why does this work? Well, for starters, the only way to *know* that you have gleaned all the information necessary from this chapter is to try to produce it just from your head. Neural networks have lots of small moving parts, and it's easy to miss one.

Why is this important for the rest of the chapters? In the following chapters, I will be referring to the concepts discussed in this chapter at a *faster pace* so that I can spend plenty of time on the newer material. It is *vitally important* that when I say something like "add your alpha parameterization to the weight update" that it is at least immediately apparent to which concepts from this chapter I'm referring.

All that is to say, memorizing small bits of neural network code has been hugely beneficial for me personally, as well as to many individuals who have taken my advice on this subject in the past.