9 Classification: Advanced Methods

In this chapter, you will learn advanced techniques for data classification. We start with **Bayesian belief networks** (Section 9.1), which unlike naïve Bayesian classifiers, do not assume class conditional independence. **Backpropagation**, a neural network algorithm, is discussed in Section 9.2. In general terms, a neural network is a set of connected input/output units in which each connection has a weight associated with it. The weights are adjusted during the learning phase to help the network predict the correct class label of the input tuples. A more recent approach to classification known as support vector machines is presented in Section 9.3. A **support vector machine** transforms training data into a higher dimension, where it finds a hyperplane that separates the data by class using essential training tuples called *support vectors*. Section 9.4 describes **classification using frequent patterns**, exploring relationships between attribute–value pairs that occur frequently in data. This methodology builds on research on frequent pattern mining (Chapters 6 and 7).

Section 9.5 presents **lazy learners** or **instance-based** methods of classification, such as nearest-neighbor classifiers and case-based reasoning classifiers, which store all of the training tuples in pattern space and wait until presented with a test tuple before performing generalization. Other approaches to classification, such as genetic algorithms, rough sets, and fuzzy logic techniques, are introduced in Section 9.6. Section 9.7 introduces additional topics in classification, including multiclass classification, semi-supervised classification, active learning, and transfer learning.

9.1 Bayesian Belief Networks

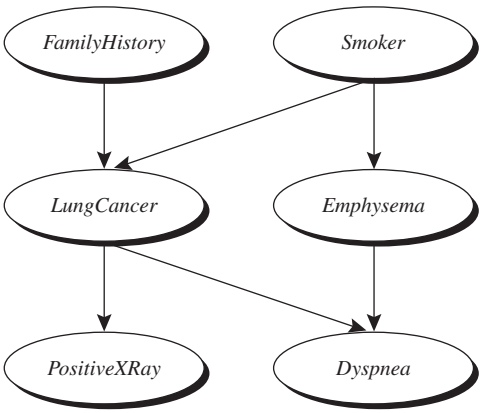
Chapter 8 introduced Bayes' theorem and naïve Bayesian classification. In this chapter, we describe *Bayesian belief networks*—probabilistic graphical models, which unlike naïve Bayesian classifiers allow the representation of dependencies among subsets of attributes. Bayesian belief networks can be used for classification. Section 9.1.1 introduces the basic concepts of Bayesian belief networks. In Section 9.1.2, you will learn how to train such models.

Data Mining: Concepts and Techniques. DOI: 10.1016/B978-0-12-381479-1.00009-5

9.1.1 Concepts and Mechanisms

The naïve Bayesian classifier makes the assumption of class conditional independence, that is, given the class label of a tuple, the values of the attributes are assumed to be conditionally independent of one another. This simplifies computation. When the assumption holds true, then the naïve Bayesian classifier is the most accurate in comparison with all other classifiers. In practice, however, dependencies can exist between variables. **Bayesian belief networks** specify joint conditional probability distributions. They allow class conditional independencies to be defined between subsets of variables. They provide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification. Bayesian belief networks are also known as **belief networks**, **Bayesian networks**, and **probabilistic networks**. For brevity, we will refer to them as belief networks.

A belief network is defined by two components—a *directed acyclic graph* and a set of *conditional probability tables* (Figure 9.1). Each node in the directed acyclic graph represents a random variable. The variables may be discrete- or continuous-valued. They may correspond to actual attributes given in the data or to "hidden variables" believed to form a relationship (e.g., in the case of medical data, a hidden variable may indicate a syndrome, representing a number of symptoms that, together, characterize a specific disease). Each arc represents a probabilistic dependence. If an arc is drawn from a node Y to a node Z, then Y is a **parent** or **immediate predecessor** of Z, and Z is a **descendant**



(a)

	FH, S	*FH, ~S*	*~FH, S*	*~FH, ~S*
LC	0.8	0.5	0.7	0.1
~LC	0.2	0.5	0.3	0.9

(b)

Figure 9.1 Simple Bayesian belief network. (a) A proposed causal model, represented by a directed acyclic graph. (b) The conditional probability table for the values of the variable *LungCancer* (*LC*) showing each possible combination of the values of its parent nodes, *FamilyHistory* (*FH*) and *Smoker* (*S*). *Source:* Adapted from Russell, Binder, Koller, and Kanazawa [RBKK95].

of Y. *Each variable is conditionally independent of its nondescendants in the graph, given its parents.*

Figure 9.1 is a simple belief network, adapted from Russell, Binder, Koller, and Kanazawa [RBKK95] for six Boolean variables. The arcs in Figure 9.1(a) allow a representation of causal knowledge. For example, having lung cancer is influenced by a person's family history of lung cancer, as well as whether or not the person is a smoker. Note that the variable *PositiveXRay* is independent of whether the patient has a family history of lung cancer or is a smoker, given that we know the patient has lung cancer. In other words, once we know the outcome of the variable *LungCancer*, then the variables *FamilyHistory* and *Smoker* do not provide any additional information regarding *PositiveXRay*. The arcs also show that the variable *LungCancer* is conditionally independent of *Emphysema*, given its parents, *FamilyHistory* and *Smoker*.

A belief network has one **conditional probability table (CPT)** for each variable. The CPT for a variable Y specifies the conditional distribution $P(Y|Parents(Y))$, where $Parents(Y)$ are the parents of Y. Figure 9.1(b) shows a CPT for the variable *LungCancer*. The conditional probability for each known value of *LungCancer* is given for each possible combination of the values of its parents. For instance, from the upper leftmost and bottom rightmost entries, respectively, we see that

$$P(LungCancer = yes \,|\, FamilyHistory = yes,\ Smoker = yes) = 0.8$$
$$P(LungCancer = no \,|\, FamilyHistory = no,\ Smoker = no) = 0.9.$$

Let $\boldsymbol{X} = (x_1, \ldots, x_n)$ be a data tuple described by the variables or attributes $Y_1, \ldots, Y_n$, respectively. Recall that each variable is conditionally independent of its nondescendants in the network graph, given its parents. This allows the network to provide a complete representation of the existing joint probability distribution with the following equation:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|Parents(Y_i)), \tag{9.1}$$

where $P(x_1, \ldots, x_n)$ is the probability of a particular combination of values of $\boldsymbol{X}$, and the values for $P(x_i|Parents(Y_i))$ correspond to the entries in the CPT for Y_i.

A node within the network can be selected as an "output" node, representing a class label attribute. There may be more than one output node. Various algorithms for inference and learning can be applied to the network. Rather than returning a single class label, the classification process can return a probability distribution that gives the probability of each class. Belief networks can be used to answer probability of evidence queries (e.g., what is the probability that an individual will have *LungCancer*, given that they have both *PositiveXRay* and *Dyspnea*) and most probable explanation queries (e.g., which group of the population is most likely to have both *PositiveXRay* and *Dyspnea*).

Belief networks have been used to model a number of well-known problems. One example is genetic linkage analysis (e.g., the mapping of genes onto a chromosome). By casting the gene linkage problem in terms of inference on Bayesian networks, and using

state-of-the art algorithms, the scalability of such analysis has advanced considerably. Other applications that have benefited from the use of belief networks include computer vision (e.g., image restoration and stereo vision), document and text analysis, decision-support systems, and sensitivity analysis. The ease with which many applications can be reduced to Bayesian network inference is advantageous in that it curbs the need to invent specialized algorithms for each such application.

9.1.2 Training Bayesian Belief Networks

"*How does a Bayesian belief network learn?*" In the learning or training of a belief network, a number of scenarios are possible. The network **topology** (or "layout" of nodes and arcs) may be constructed by human experts or inferred from the data. The network variables may be *observable* or *hidden* in all or some of the training tuples. The hidden data case is also referred to as *missing values* or *incomplete data*.

Several algorithms exist for learning the network topology from the training data given observable variables. The problem is one of discrete optimization. For solutions, please see the bibliographic notes at the end of this chapter (Section 9.10). Human experts usually have a good grasp of the direct conditional dependencies that hold in the domain under analysis, which helps in network design. Experts must specify conditional probabilities for the nodes that participate in direct dependencies. These probabilities can then be used to compute the remaining probability values.

If the network topology is known and the variables are observable, then training the network is straightforward. It consists of computing the CPT entries, as is similarly done when computing the probabilities involved in naïve Bayesian classification.

When the network topology is given and some of the variables are hidden, there are various methods to choose from for training the belief network. We will describe a promising method of gradient descent. For those without an advanced math background, the description may look rather intimidating with its calculus-packed formulae. However, packaged software exists to solve these equations, and the general idea is easy to follow.

Let D be a training set of data tuples, $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_{|D|}$. Training the belief network means that we must learn the values of the CPT entries. Let w_{ijk} be a CPT entry for the variable $Y_i = y_{ij}$ having the parents $U_i = u_{ik}$, where $w_{ijk} \equiv P(Y_i = y_{ij} | U_i = u_{ik})$. For example, if w_{ijk} is the upper leftmost CPT entry of Figure 9.1(b), then Y_i is *LungCancer*; y_{ij} is its value, "*yes*"; U_i lists the parent nodes of Y_i, namely, {*FamilyHistory, Smoker*}; and u_{ik} lists the values of the parent nodes, namely, {"*yes*", "*yes*"}. The w_{ijk} are viewed as weights, analogous to the weights in hidden units of neural networks (Section 9.2). The set of weights is collectively referred to as $\boldsymbol{W}$. The weights are initialized to random probability values. A *gradient descent* strategy performs greedy hill-climbing. At each iteration, the weights are updated and will eventually converge to a local optimum solution.

A **gradient descent** strategy is used to search for the w_{ijk} values that best model the data, based on the assumption that each possible setting of w_{ijk} is equally likely. Such

a strategy is iterative. It searches for a solution along the negative of the gradient (i.e., steepest descent) of a criterion function. We want to find the set of weights, $\boldsymbol{W}$, that maximize this function. To start with, the weights are initialized to random probability values. The gradient descent method performs greedy hill-climbing in that, at each iteration or step along the way, the algorithm moves toward what appears to be the best solution at the moment, without backtracking. The weights are updated at each iteration. Eventually, they converge to a local optimum solution.

For our problem, we maximize $P_w(D) = \prod_{d=1}^{|D|} P_w(\boldsymbol{X}_d)$. This can be done by following the gradient of $\ln P_w(S)$, which makes the problem simpler. Given the network topology and initialized w_{ijk}, the algorithm proceeds as follows:

1. **Compute the gradients:** For each i, j, k, compute

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}} = \sum_{d=1}^{|D|} \frac{P(Y_i = y_{ij}, U_i = u_{ik} | \boldsymbol{X}_d)}{w_{ijk}}. \tag{9.2}$$

   The probability on the right side of Eq. (9.2) is to be calculated for each training tuple, $\boldsymbol{X}_d$, in D. For brevity, let's refer to this probability simply as p. When the variables represented by Y_i and U_i are hidden for some $\boldsymbol{X}_d$, then the corresponding probability p can be computed from the observed variables of the tuple using standard algorithms for Bayesian network inference such as those available in the commercial software package HUGIN (*www.hugin.dk*).

2. **Take a small step in the direction of the gradient:** The weights are updated by

$$w_{ijk} \leftarrow w_{ijk} + (l) \frac{\partial \ln P_w(D)}{\partial w_{ijk}}, \tag{9.3}$$

   where l is the **learning rate** representing the step size and $\frac{\partial \ln P_w(D)}{\partial w_{ijk}}$ is computed from Eq. (9.2). The learning rate is set to a small constant and helps with convergence.

3. **Renormalize the weights:** Because the weights w_{ijk} are probability values, they must be between 0.0 and 1.0, and $\sum_j w_{ijk}$ must equal 1 for all i, k. These criteria are achieved by renormalizing the weights after they have been updated by Eq. (9.3).

Algorithms that follow this learning form are called *adaptive probabilistic networks*. Other methods for training belief networks are referenced in the bibliographic notes at the end of this chapter (Section 9.10). Belief networks are computationally intensive. Because belief networks provide explicit representations of causal structure, a human expert can provide prior knowledge to the training process in the form of network topology and/or conditional probability values. This can significantly improve the learning rate.

9.2 Classification by Backpropagation

"*What is backpropagation?*" Backpropagation is a neural network learning algorithm. The neural networks field was originally kindled by psychologists and neurobiologists who sought to develop and test computational analogs of neurons. Roughly speaking, a **neural network** is a set of connected input/output units in which each connection has a weight associated with it. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples. Neural network learning is also referred to as **connectionist learning** due to the connections between units.

Neural networks involve long training times and are therefore more suitable for applications where this is feasible. They require a number of parameters that are typically best determined empirically such as the network topology or "structure." Neural networks have been criticized for their poor interpretability. For example, it is difficult for humans to interpret the symbolic meaning behind the learned weights and of "hidden units" in the network. These features initially made neural networks less desirable for data mining.

Advantages of neural networks, however, include their high tolerance of noisy data as well as their ability to classify patterns on which they have not been trained. They can be used when you may have little knowledge of the relationships between attributes and classes. They are well suited for continuous-valued inputs *and* outputs, unlike most decision tree algorithms. They have been successful on a wide array of real-world data, including handwritten character recognition, pathology and laboratory medicine, and training a computer to pronounce English text. Neural network algorithms are inherently parallel; parallelization techniques can be used to speed up the computation process. In addition, several techniques have been recently developed for rule extraction from trained neural networks. These factors contribute to the usefulness of neural networks for classification and numeric prediction in data mining.

There are many different kinds of neural networks and neural network algorithms. The most popular neural network algorithm is *backpropagation*, which gained repute in the 1980s. In Section 9.2.1 you will learn about multilayer feed-forward networks, the type of neural network on which the backpropagation algorithm performs. Section 9.2.2 discusses defining a network topology. The backpropagation algorithm is described in Section 9.2.3. Rule extraction from trained neural networks is discussed in Section 9.2.4.

9.2.1 A Multilayer Feed-Forward Neural Network

The backpropagation algorithm performs learning on a *multilayer feed-forward* neural network. It iteratively learns a set of weights for prediction of the class label of tuples. A **multilayer feed-forward** neural network consists of an *input layer*, one or more *hidden layers*, and an *output layer*. An example of a multilayer feed-forward network is shown in Figure 9.2.

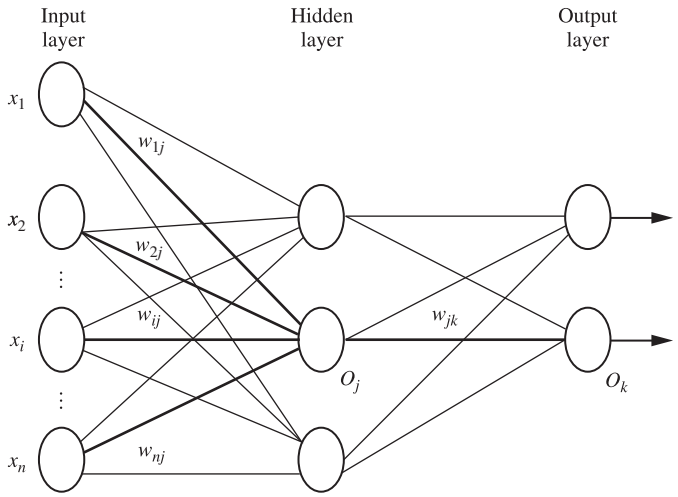


Figure 9.2 Multilayer feed-forward neural network.

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

The units in the input layer are called **input units**. The units in the hidden layers and output layer are sometimes referred to as **neurodes**, due to their symbolic biological basis, or as **output units**. The multilayer neural network shown in Figure 9.2 has two layers of output units. Therefore, we say that it is a **two-layer** neural network. (The input layer is not counted because it serves only to pass the input values to the next layer.) Similarly, a network containing two hidden layers is called a *three-layer* neural network, and so on. It is a feed-forward network since none of the weights cycles back to an input unit or to a previous layer's output unit. It is **fully connected** in that each unit provides input to each unit in the next forward layer.

Each output unit takes, as input, a weighted sum of the outputs from units in the previous layer (see Figure 9.4 later). It applies a nonlinear (activation) function to the weighted input. Multilayer feed-forward neural networks are able to model the class prediction as a nonlinear combination of the inputs. From a statistical point of view, they perform nonlinear regression. *Multilayer feed-forward networks, given enough hidden units and enough training samples, can closely approximate any function.*

9.2.2 Defining a Network Topology

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

Normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. Typically, input values are normalized so as to fall between 0.0 and 1.0. Discrete-valued attributes may be encoded such that there is one input unit per domain value. For example, if an attribute A has three possible or known values, namely $\{a_0, a_1, a_2\}$, then we may assign three input units to represent A. That is, we may have, say, I_0, I_1, I_2 as input units. Each unit is initialized to 0. If $A = a_0$, then I_0 is set to 1 and the rest are 0. If $A = a_1$, then I_1 is set to 1 and the rest are 0, and so on.

Neural networks can be used for both classification (to predict the class label of a given tuple) and numeric prediction (to predict a continuous-valued output). For classification, one output unit may be used to represent two classes (where the value 1 represents one class, and the value 0 represents the other). If there are more than two classes, then one output unit per class is used. (See Section 9.7.1 for more strategies on multiclass classification.)

There are no clear rules as to the "best" number of hidden layer units. Network design is a trial-and-error process and may affect the accuracy of the resulting trained network. The initial values of the weights may also affect the resulting accuracy. Once a network has been trained and its accuracy is not considered acceptable, it is common to repeat the training process with a different network topology or a different set of initial weights. Cross-validation techniques for accuracy estimation (described in Chapter 8) can be used to help decide when an acceptable network has been found. A number of automated techniques have been proposed that search for a "good" network structure. These typically use a hill-climbing approach that starts with an initial structure that is selectively modified.

9.2.3 Backpropagation

"*How does backpropagation work?*" Backpropagation learns by iteratively processing a data set of training tuples, comparing the network's prediction for each tuple with the actual known *target* value. The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for numeric prediction). For each training tuple, the weights are modified so as to minimize the mean-squared error between the network's prediction and the actual target value. These modifications are made in the "backwards" direction (i.e., from the output layer) through each hidden layer down to the first hidden layer (hence the name *backpropagation*). Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops. The algorithm is summarized in Figure 9.3. The steps involved are expressed in terms of inputs, outputs, and errors, and may seem awkward if this is your first look at

Algorithm: Backpropagation. Neural network learning for classification or numeric prediction, using the backpropagation algorithm.
Input:

- D, a data set consisting of the training tuples and their associated target values;
- l, the learning rate;
- *network*, a multilayer feed-forward network.

Output: A trained neural network.
Method:

(1) Initialize all weights and biases in *network*;
(2) **while** terminating condition is not satisfied {
(3) **for** each training tuple $\boldsymbol{X}$ in D {
(4) // Propagate the inputs forward:
(5) **for** each input layer unit j {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7) **for** each hidden or output layer unit j {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to the previous layer, i
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10) // Backpropagate the errors:
(11) **for** each unit j in the output layer
(12) $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13) **for** each unit j in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
(15) **for** each weight w_{ij} in *network* {
(16) $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18) **for** each bias θ_j in *network* {
(19) $\Delta \theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21) } }

Figure 9.3 Backpropagation algorithm.

neural network learning. However, once you become familiar with the process, you will see that each step is inherently simple. The steps are described next.

Initialize the weights: The weights in the network are initialized to small random numbers (e.g., ranging from -1.0 to 1.0, or -0.5 to 0.5). Each unit has a *bias* associated with it, as explained later. The biases are similarly initialized to small random numbers.

Each training tuple, $\boldsymbol{X}$, is processed by the following steps.

Propagate the inputs forward: First, the training tuple is fed to the network's input layer. The inputs pass through the input units, unchanged. That is, for an input unit, j,

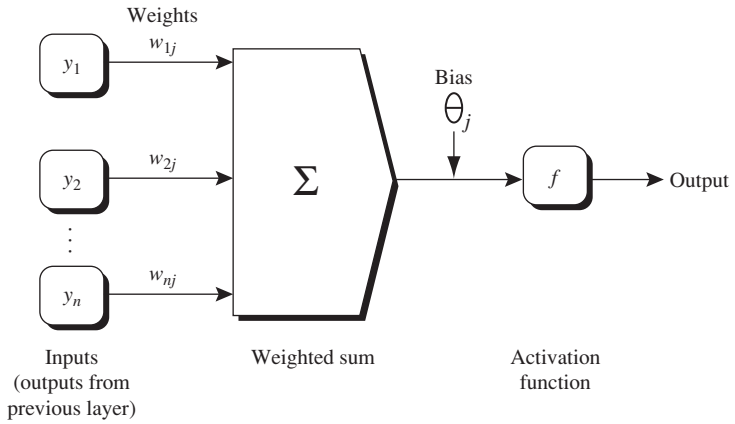


Figure 9.4 Hidden or output layer unit *j*: The inputs to unit *j* are outputs from the previous layer. These are multiplied by their corresponding weights to form a weighted sum, which is added to the bias associated with unit *j*. A nonlinear activation function is applied to the net input. (For ease of explanation, the inputs to unit *j* are labeled $y_1, y_2, \ldots, y_n$. If unit *j* were in the first hidden layer, then these inputs would correspond to the input tuple $(x_1, x_2, \ldots, x_n)$.)

its output, O_j, is equal to its input value, I_j. Next, the net input and output of each unit in the hidden and output layers are computed. The net input to a unit in the hidden or output layers is computed as a linear combination of its inputs. To help illustrate this point, a hidden layer or output layer unit is shown in Figure 9.4. Each such unit has a number of inputs to it that are, in fact, the outputs of the units connected to it in the previous layer. Each connection has a weight. To compute the net input to the unit, each input connected to the unit is multiplied by its corresponding weight, and this is summed. Given a unit, *j* in a hidden or output layer, the net input, I_j, to unit *j* is

$$I_j = \sum_i w_{ij} O_i + \theta_j, \tag{9.4}$$

where w_{ij} is the weight of the connection from unit *i* in the previous layer to unit *j*; O_i is the output of unit *i* from the previous layer; and θ_j is the **bias** of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

Each unit in the hidden and output layers takes its net input and then applies an **activation** function to it, as illustrated in Figure 9.4. The function symbolizes the activation of the neuron represented by the unit. The **logistic**, or **sigmoid**, function is used. Given the net input I_j to unit *j*, then O_j, the output of unit *j*, is computed as

$$O_j = \frac{1}{1 + e^{-I_j}}. \tag{9.5}$$

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

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

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

$$Err_j = O_j(1 - O_j)(T_j - O_j), \tag{9.6}$$

where O_j is the actual output of unit j, and T_j is the known target value of the given training tuple. Note that $O_j(1 - O_j)$ is the derivative of the logistic function.

To compute the error of a hidden layer unit j, the weighted sum of the errors of the units connected to unit j in the next layer are considered. The error of a hidden layer unit j is

$$Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}, \tag{9.7}$$

where w_{jk} is the weight of the connection from unit j to a unit k in the next higher layer, and Err_k is the error of unit k.

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

$$\Delta w_{ij} = (l) Err_j O_i. \tag{9.8}$$

$$w_{ij} = w_{ij} + \Delta w_{ij}. \tag{9.9}$$

"*What is* l *in Eq. (9.8)?*" The variable l is the **learning rate**, a constant typically having a value between 0.0 and 1.0. Backpropagation learns using a gradient descent method to search for a set of weights that fits the training data so as to minimize the mean-squared distance between the network's class prediction and the known target value of the tuples.[1] The learning rate helps avoid getting stuck at a local minimum in decision space (i.e., where the weights appear to converge, but are not the optimum solution) and encourages finding the global minimum. If the learning rate is too small, then learning will occur at a very slow pace. If the learning rate is too large, then oscillation between

[1] A method of gradient descent was also used for training Bayesian belief networks, as described in Section 9.1.2.

inadequate solutions may occur. A rule of thumb is to set the learning rate to $1/t$, where t is the number of iterations through the training set so far.

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

$$\Delta\theta_j = (l)Err_j. \tag{9.10}$$

$$\theta_j = \theta_j + \Delta\theta_j. \tag{9.11}$$

Note that here we are updating the weights and biases after the presentation of each tuple. This is referred to as **case updating**. Alternatively, the weight and bias increments could be accumulated in variables, so that the weights and biases are updated after all the tuples in the training set have been presented. This latter strategy is called **epoch updating**, where one iteration through the training set is an **epoch**. In theory, the mathematical derivation of backpropagation employs epoch updating, yet in practice, case updating is more common because it tends to yield more accurate results.

Terminating condition: Training stops when

- All Δw_{ij} in the previous epoch are so small as to be below some specified threshold, or
- The percentage of tuples misclassified in the previous epoch is below some threshold, or
- A prespecified number of epochs has expired.

In practice, several hundreds of thousands of epochs may be required before the weights will converge.

"*How efficient is backpropagation?*" The computational efficiency depends on the time spent training the network. Given $|D|$ tuples and w weights, each epoch requires $O(|D| \times w)$ time. However, in the worst-case scenario, the number of epochs can be exponential in n, the number of inputs. In practice, the time required for the networks to converge is highly variable. A number of techniques exist that help speed up the training time. For example, a technique known as *simulated annealing* can be used, which also ensures convergence to a global optimum.

Example 9.1 Sample calculations for learning by the backpropagation algorithm. Figure 9.5 shows a multilayer feed-forward neural network. Let the learning rate be 0.9. The initial weight and bias values of the network are given in Table 9.1, along with the first training tuple, $\boldsymbol{X} = (1, 0, 1)$, with a class label of 1.

This example shows the calculations for backpropagation, given the first training tuple, $\boldsymbol{X}$. The tuple is fed into the network, and the net input and output of each unit

are computed. These values are shown in Table 9.2. The error of each unit is computed and propagated backward. The error values are shown in Table 9.3. The weight and bias updates are shown in Table 9.4. ■

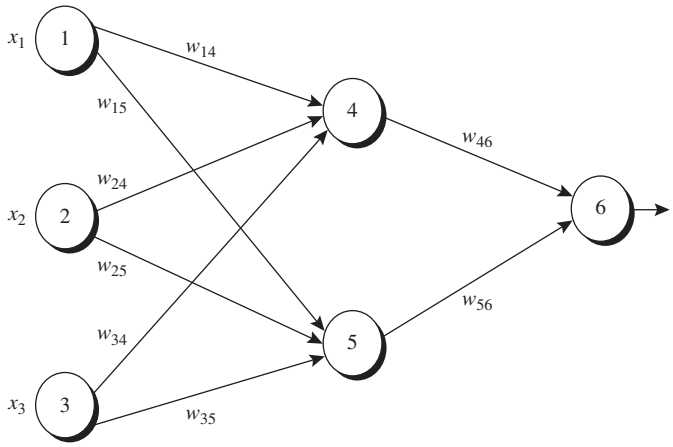


Figure 9.5 Example of a multilayer feed-forward neural network.

Table 9.1 Initial Input, Weight, and Bias Values

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

Table 9.2 Net Input and Output Calculations

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

Table 9.3 Calculation of the Error at Each Node

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

Table 9.4 Calculations for Weight and Bias Updating

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

"*How can we classify an unknown tuple using a trained network?*" To classify an unknown tuple, $\boldsymbol{X}$, the tuple is input to the trained network, and the net input and output of each unit are computed. (There is no need for computation and/or backpropagation of the error.) If there is one output node per class, then the output node with the highest value determines the predicted class label for $\boldsymbol{X}$. If there is only one output node, then output values greater than or equal to 0.5 may be considered as belonging to the positive class, while values less than 0.5 may be considered negative.

Several variations and alternatives to the backpropagation algorithm have been proposed for classification in neural networks. These may involve the dynamic adjustment of the network topology and of the learning rate or other parameters, or the use of different error functions.

9.2.4 Inside the Black Box: Backpropagation and Interpretability

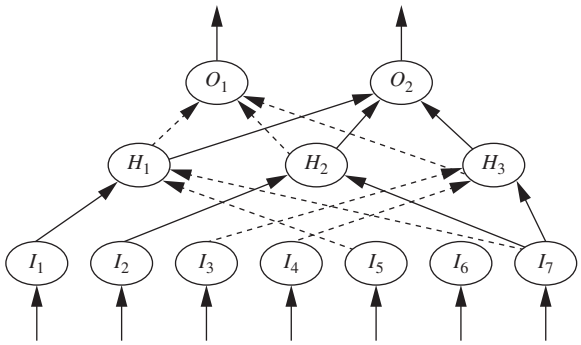
"*Neural networks are like a black box. How can I 'understand' what the backpropagation network has learned?*" A major disadvantage of neural networks lies in their knowledge representation. Acquired knowledge in the form of a network of units connected by weighted links is difficult for humans to interpret. This factor has motivated research in extracting the knowledge embedded in trained neural networks and in representing that knowledge symbolically. Methods include extracting rules from networks and sensitivity analysis.

Various algorithms for rule extraction have been proposed. The methods typically impose restrictions regarding procedures used in training the given neural network, the network topology, and the discretization of input values.

Fully connected networks are difficult to articulate. Hence, often the first step in extracting rules from neural networks is **network pruning**. This consists of simplifying

the network structure by removing weighted links that have the least effect on the trained network. For example, a weighted link may be deleted if such removal does not result in a decrease in the classification accuracy of the network.

Once the trained network has been pruned, some approaches will then perform link, unit, or activation value clustering. In one method, for example, clustering is used to find the set of common activation values for each hidden unit in a given trained two-layer neural network (Figure 9.6). The combinations of these activation values for each hidden unit are analyzed. Rules are derived relating combinations of activation values



Identify sets of common activation values for each hidden node, H_i:
for H_1: (–1,0,1)
for H_2: (0,1)
for H_3: (–1,0.24,1)

Derive rules relating common activation values with output nodes, O_j:
IF (H_2=0 AND H_3=–1) OR
(H_1=–1 AND H_2=1 AND H_3=–1) OR
(H_1=–1 AND H_2=0 AND H_3=0.24)
THEN O_1=1, O_2=0
ELSE O_1=0, O_2=1

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

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

Figure 9.6 Rules can be extracted from training neural networks. *Source:* Adapted from Lu, Setiono, and Liu [LSL95].

with corresponding output unit values. Similarly, the sets of input values and activation values are studied to derive rules describing the relationship between the input layer and the hidden "layer units"? Finally, the two sets of rules may be combined to form IF-THEN rules. Other algorithms may derive rules of other forms, including *M*-of-*N* rules (where *M* out of a given *N* conditions in the rule antecedent must be true for the rule consequent to be applied), decision trees with *M*-of-*N* tests, fuzzy rules, and finite automata.

Sensitivity analysis is used to assess the impact that a given input variable has on a network output. The input to the variable is varied while the remaining input variables are fixed at some value. Meanwhile, changes in the network output are monitored. The knowledge gained from this analysis form can be represented in rules such as "*IF X decreases* 5% *THEN Y increases* 8%."

9.3 Support Vector Machines

In this section, we study **support vector machines (SVMs)**, a method for the classification of both linear and nonlinear data. In a nutshell, an **SVM** is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (i.e., a "decision boundary" separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using *support vectors* ("essential" training tuples) and *margins* (defined by the support vectors). We will delve more into these new concepts later.

"*I've heard that SVMs have attracted a great deal of attention lately. Why?*" The first paper on support vector machines was presented in 1992 by Vladimir Vapnik and colleagues Bernhard Boser and Isabelle Guyon, although the groundwork for SVMs has been around since the 1960s (including early work by Vapnik and Alexei Chervonenkis on statistical learning theory). Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors found also provide a compact description of the learned model. SVMs can be used for numeric prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, and speaker identification, as well as benchmark time-series prediction tests.

9.3.1 The Case When the Data Are Linearly Separable

To explain the mystery of SVMs, let's first look at the simplest case—a two-class problem where the classes are linearly separable. Let the data set D be given as $(\boldsymbol{X}_1, y_1)$, $(\boldsymbol{X}_2, y_2), \ldots, (\boldsymbol{X}_{|D|}, y_{|D|})$, where $\boldsymbol{X}_i$ is the set of training tuples with associated class labels, y_i. Each y_i can take one of two values, either $+1$ or -1 (i.e., $y_i \in \{+1, -1\}$),

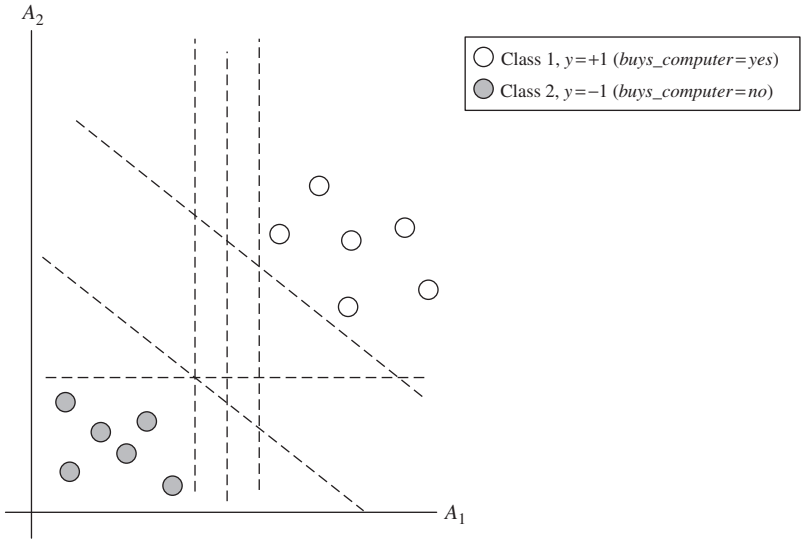


Figure 9.7 The 2-D training data are linearly separable. There are an infinite number of possible separating hyperplanes or "decision boundaries," some of which are shown here as dashed lines. Which one is best?

corresponding to the classes *buys_computer* = *yes* and *buys_computer* = *no*, respectively. To aid in visualization, let's consider an example based on two input attributes, A_1 and A_2, as shown in Figure 9.7. From the graph, we see that the 2-D data are **linearly separable** (or "linear," for short), because a straight line can be drawn to separate all the tuples of class $+1$ from all the tuples of class -1.

There are an infinite number of separating lines that could be drawn. We want to find the "best" one, that is, one that (we hope) will have the minimum classification error on previously unseen tuples. How can we find this best line? Note that if our data were 3-D (i.e., with three attributes), we would want to find the best separating *plane*. Generalizing to n dimensions, we want to find the best *hyperplane*. We will use "hyperplane" to refer to the decision boundary that we are seeking, regardless of the number of input attributes. So, in other words, how can we find the best hyperplane?

An SVM approaches this problem by searching for the **maximum marginal hyperplane**. Consider Figure 9.8, which shows two possible separating hyperplanes and their associated margins. Before we get into the definition of margins, let's take an intuitive look at this figure. Both hyperplanes can correctly classify all the given data tuples. Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase) the SVM searches for the hyperplane with the largest margin, that is, the *maximum marginal hyperplane* (MMH). The associated margin gives the largest separation between classes.

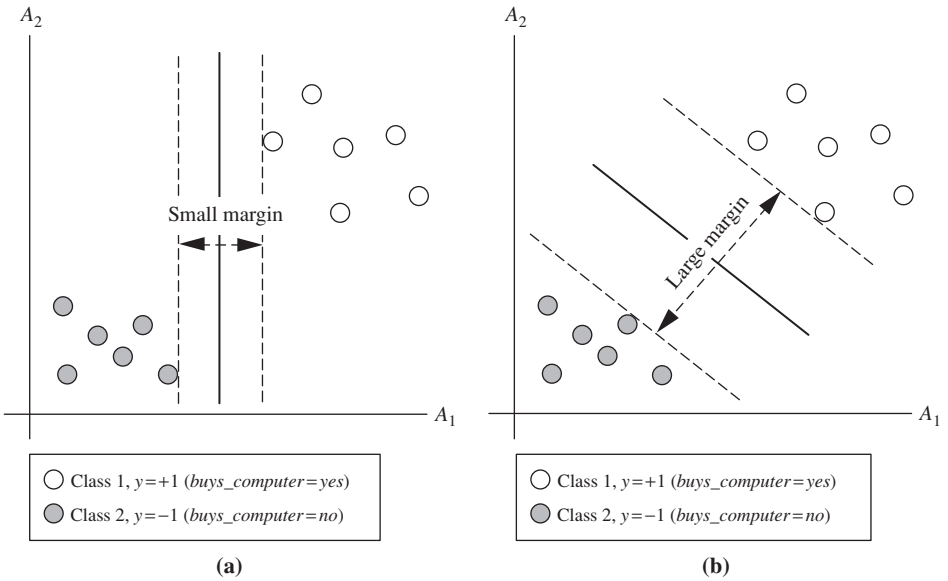


Figure 9.8 Here we see just two possible separating hyperplanes and their associated margins. Which one is better? The one with the larger margin (b) should have greater generalization accuracy.

Getting to an informal definition of **margin**, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the "sides" of the margin are parallel to the hyperplane. When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class.

A separating hyperplane can be written as

$$\boldsymbol{W} \cdot \boldsymbol{X} + b = 0, \tag{9.12}$$

where $\boldsymbol{W}$ is a weight vector, namely, $\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$; n is the number of attributes; and b is a scalar, often referred to as a bias. To aid in visualization, let's consider two input attributes, A_1 and A_2, as in Figure 9.8(b). Training tuples are 2-D (e.g., $\boldsymbol{X} = (x_1, x_2)$), where x_1 and x_2 are the values of attributes A_1 and A_2, respectively, for $\boldsymbol{X}$. If we think of b as an additional weight, w_0, we can rewrite Eq. (9.12) as

$$w_0 + w_1 x_1 + w_2 x_2 = 0. \tag{9.13}$$

Thus, any point that lies above the separating hyperplane satisfies

$$w_0 + w_1 x_1 + w_2 x_2 > 0. \tag{9.14}$$

Similarly, any point that lies below the separating hyperplane satisfies

$$w_0 + w_1 x_1 + w_2 x_2 < 0. \tag{9.15}$$

The weights can be adjusted so that the hyperplanes defining the "sides" of the margin can be written as

$$H_1 : w_0 + w_1 x_1 + w_2 x_2 \geq 1 \quad \text{for } y_i = +1, \tag{9.16}$$

$$H_2 : w_0 + w_1 x_1 + w_2 x_2 \leq -1 \quad \text{for } y_i = -1. \tag{9.17}$$

That is, any tuple that falls on or above H_1 belongs to class $+1$, and any tuple that falls on or below H_2 belongs to class -1. Combining the two inequalities of Eqs. (9.16) and (9.17), we get

$$y_i(w_0 + w_1 x_1 + w_2 x_2) \geq 1, \ \forall i. \tag{9.18}$$

Any training tuples that fall on hyperplanes H_1 or H_2 (i.e., the "sides" defining the margin) satisfy Eq. (9.18) and are called **support vectors**. That is, they are equally close to the (separating) MMH. In Figure 9.9, the support vectors are shown encircled with a thicker border. Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

From this, we can obtain a formula for the size of the maximal margin. The distance from the separating hyperplane to any point on H_1 is $\frac{1}{||\boldsymbol{W}||}$, where $||\boldsymbol{W}||$ is the Euclidean norm of $\boldsymbol{W}$, that is, $\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}}$.[2] By definition, this is equal to the distance from any point on H_2 to the separating hyperplane. Therefore, the maximal margin is $\frac{2}{||\boldsymbol{W}||}$.

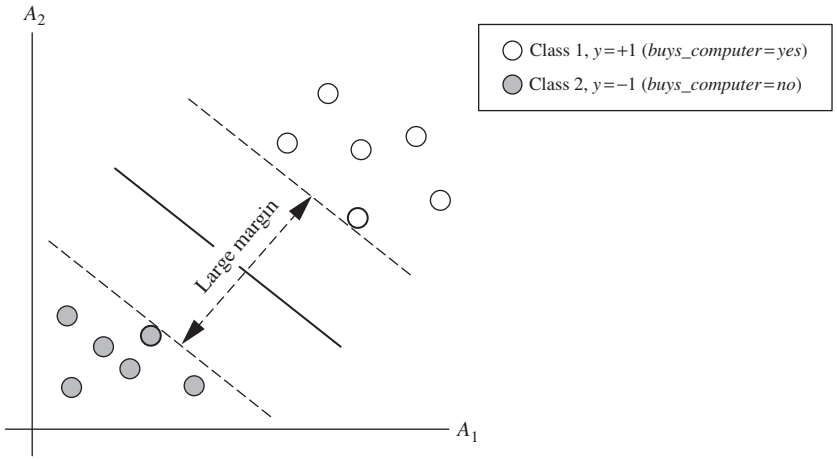


Figure 9.9 Support vectors. The SVM finds the maximum separating hyperplane, that is, the one with maximum distance between the nearest training tuples. The support vectors are shown with a thicker border.

[2] If $\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$, then $\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}} = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$.

"*So, how does an SVM find the MMH and the support vectors?*" Using some "fancy math tricks," we can rewrite Eq. (9.18) so that it becomes what is known as a constrained (convex) quadratic optimization problem. Such fancy math tricks are beyond the scope of this book. Advanced readers may be interested to note that the tricks involve rewriting Eq. (9.18) using a Lagrangian formulation and then solving for the solution using Karush-Kuhn-Tucker (KKT) conditions. Details can be found in the bibliographic notes at the end of this chapter (Section 9.10).

If the data are small (say, less than 2000 training tuples), any optimization software package for solving constrained convex quadratic problems can then be used to find the support vectors and MMH. For larger data, special and more efficient algorithms for training SVMs can be used instead, the details of which exceed the scope of this book. Once we've found the support vectors and MMH (note that the support vectors define the MMH!), we have a trained support vector machine. The MMH is a linear class boundary, and so the corresponding SVM can be used to classify linearly separable data. We refer to such a trained SVM as a *linear SVM*.

"*Once I've got a trained support vector machine, how do I use it to classify test (i.e., new) tuples?*" Based on the Lagrangian formulation mentioned before, the MMH can be rewritten as the decision boundary

$$d(\boldsymbol{X}^T) = \sum_{i=1}^{l} y_i \alpha_i \boldsymbol{X}_i \boldsymbol{X}^T + b_0, \tag{9.19}$$

where y_i is the class label of support vector $\boldsymbol{X}_i$; $\boldsymbol{X}^T$ is a test tuple; α_i and b_0 are numeric parameters that were determined automatically by the optimization or SVM algorithm noted before; and l is the number of support vectors.

Interested readers may note that the α_i are Lagrangian multipliers. For linearly separable data, the support vectors are a subset of the actual training tuples (although there will be a slight twist regarding this when dealing with nonlinearly separable data, as we shall see in the following).

Given a test tuple, $\boldsymbol{X}^T$, we plug it into Eq. (9.19), and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then $\boldsymbol{X}^T$ falls on or above the MMH, and so the SVM predicts that $\boldsymbol{X}^T$ belongs to class $+1$ (representing *buys_computer* $=$ *yes*, in our case). If the sign is negative, then $\boldsymbol{X}^T$ falls on or below the MMH and the class prediction is -1 (representing *buys_computer* $=$ *no*).

Notice that the Lagrangian formulation of our problem (Eq. 9.19) contains a dot product between support vector $\boldsymbol{X}_i$ and test tuple $\boldsymbol{X}^T$. This will prove very useful for finding the MMH and support vectors for the case when the given data are nonlinearly separable, as described further in the next section.

Before we move on to the nonlinear case, there are two more important things to note. The complexity of the learned classifier is characterized by the number of support vectors rather than the dimensionality of the data. Hence, SVMs tend to be less prone to overfitting than some other methods. The support vectors are the essential or critical training tuples—they lie closest to the decision boundary (MMH). If all other training

tuples were removed and training were repeated, the same separating hyperplane would be found. Furthermore, the number of support vectors found can be used to compute an (upper) bound on the expected error rate of the SVM classifier, which is independent of the data dimensionality. An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high.

9.3.2 The Case When the Data Are Linearly Inseparable

In Section 9.3.1 we learned about linear SVMs for classifying linearly separable data, but what if the data are not linearly separable, as in Figure 9.10? In such cases, no straight line can be found that would separate the classes. The linear SVMs we studied would not be able to find a feasible solution here. Now what?

The good news is that the approach described for linear SVMs can be extended to create *nonlinear SVMs* for the classification of *linearly inseparable data* (also called *nonlinearly separable data*, or *nonlinear data* for short). Such SVMs are capable of finding nonlinear decision boundaries (i.e., nonlinear hypersurfaces) in input space.

"*So*," you may ask, "*how can we extend the linear approach?*" We obtain a nonlinear SVM by extending the approach for linear SVMs as follows. There are two main steps. In the first step, we transform the original input data into a higher dimensional space using a nonlinear mapping. Several common nonlinear mappings can be used in this step, as we will further describe next. Once the data have been transformed into the new higher space, the second step searches for a linear separating hyperplane in the new space. We again end up with a quadratic optimization problem that can be solved using the linear SVM formulation. The maximal marginal hyperplane found in the new space corresponds to a nonlinear separating hypersurface in the original space.

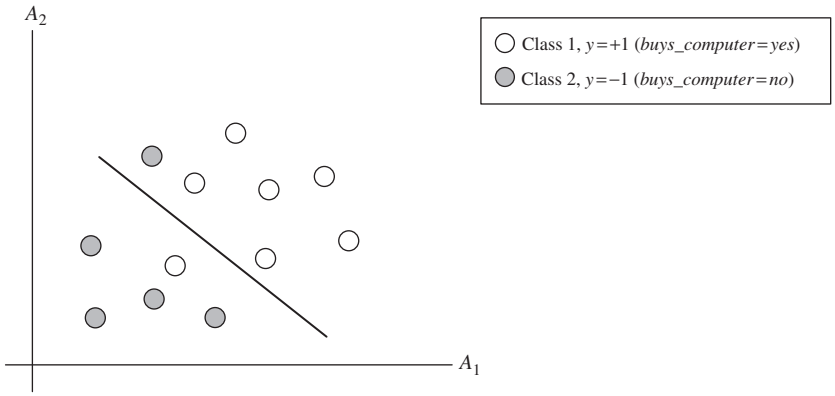


Figure 9.10 A simple 2-D case showing linearly inseparable data. Unlike the linear separable data of Figure 9.7, here it is not possible to draw a straight line to separate the classes. Instead, the decision boundary is nonlinear.

Example 9.2 **Nonlinear transformation of original input data into a higher dimensional space.** Consider the following example. A 3-D input vector $\boldsymbol{X} = (x_1, x_2, x_3)$ is mapped into a 6-D space, $\boldsymbol{Z}$, using the mappings $\phi_1(\boldsymbol{X}) = x_1$, $\phi_2(\boldsymbol{X}) = x_2$, $\phi_3(\boldsymbol{X}) = x_3$, $\phi_4(\boldsymbol{X}) = (x_1)^2$, $\phi_5(\boldsymbol{X}) = x_1x_2$, and $\phi_6(\boldsymbol{X}) = x_1x_3$. A decision hyperplane in the new space is $d(\boldsymbol{Z}) = \boldsymbol{WZ} + b$, where $\boldsymbol{W}$ and $\boldsymbol{Z}$ are vectors. This is linear. We solve for $\boldsymbol{W}$ and b and then substitute back so that the linear decision hyperplane in the new ($\boldsymbol{Z}$) space corresponds to a nonlinear second-order polynomial in the original 3-D input space:

$$\begin{aligned} d(\boldsymbol{Z}) &= w_1x_1 + w_2x_2 + w_3x_3 + w_4(x_1)^2 + w_5x_1x_2 + w_6x_1x_3 + b \\ &= w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 + w_6z_6 + b. \end{aligned}$$

■

But there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Refer to Eq. (9.19) for the classification of a test tuple, $\boldsymbol{X^T}$. Given the test tuple, we have to compute its dot product with every one of the support vectors.[3] In training, we have to compute a similar dot product several times in order to find the MMH. This is especially expensive. Hence, the dot product computation required is very heavy and costly. We need another trick!

Luckily, we can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot products, $\phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j)$, where $\phi(\boldsymbol{X})$ is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead apply a *kernel function*, $K(\boldsymbol{X}_i, \boldsymbol{X}_j)$, to the original input data. That is,

$$K(\boldsymbol{X}_i, \boldsymbol{X}_j) = \phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j). \tag{9.20}$$

In other words, everywhere that $\phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j)$ appears in the training algorithm, we can replace it with $K(\boldsymbol{X}_i, \boldsymbol{X}_j)$. In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don't even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem.

After applying this trick, we can then proceed to find a maximal separating hyperplane. The procedure is similar to that described in Section 9.3.1, although it involves placing a user-specified upper bound, C, on the Lagrange multipliers, α_i. This upper bound is best determined experimentally.

"*What are some of the kernel functions that could be used?*" Properties of the kinds of kernel functions that could be used to replace the dot product scenario just described

[3]The dot product of two vectors, $\boldsymbol{X}^T = (x_1^T, x_2^T, \ldots, x_n^T)$ and $\boldsymbol{X}_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ is $x_1^T x_{i1} + x_2^T x_{i2} + \cdots + x_n^T x_{in}$. Note that this involves one multiplication and one addition for each of the n dimensions.

have been studied. Three admissible kernel functions are

$$\textbf{Polynomial kernel of degree } \boldsymbol{h}\textbf{:} \quad K(\boldsymbol{X}_i, \boldsymbol{X}_j) = (\boldsymbol{X}_i \cdot \boldsymbol{X}_j + 1)^h$$

$$\textbf{Gaussian radial basis function kernel:} \quad K(\boldsymbol{X}_i, \boldsymbol{X}_j) = e^{-\|\boldsymbol{X}_i - \boldsymbol{X}_j\|^2/2\sigma^2}$$

$$\textbf{Sigmoid kernel:} \quad K(\boldsymbol{X}_i, \boldsymbol{X}_j) = \tanh(\kappa \boldsymbol{X}_i \cdot \boldsymbol{X}_j - \delta)$$

Each of these results in a different nonlinear classifier in (the original) input space. Neural network aficionados will be interested to note that the resulting decision hyperplanes found for nonlinear SVMs are the same type as those found by other well-known neural network classifiers. For instance, an SVM with a Gaussian radial basis function (RBF) gives the same decision hyperplane as a type of neural network known as a radial basis function network. An SVM with a sigmoid kernel is equivalent to a simple two-layer neural network known as a multilayer perceptron (with no hidden layers).

There are no golden rules for determining which admissible kernel will result in the most accurate SVM. In practice, the kernel chosen does not generally make a large difference in resulting accuracy. SVM training always finds a global solution, unlike neural networks, such as backpropagation, where many local minima usually exist (Section 9.2.3).

So far, we have described linear and nonlinear SVMs for binary (i.e., two-class) classification. SVM classifiers can be combined for the multiclass case. See Section 9.7.1 for some strategies, such as training one classifier per class and the use of error-correcting codes.

A major research goal regarding SVMs is to improve the speed in training and testing so that SVMs may become a more feasible option for very large data sets (e.g., millions of support vectors). Other issues include determining the best kernel for a given data set and finding more efficient methods for the multiclass case.

9.4 Classification Using Frequent Patterns

Frequent patterns show interesting relationships between attribute–value pairs that occur frequently in a given data set. For example, we may find that the attribute–value pairs *age* = *youth* and *credit* = *OK* occur in 20% of data tuples describing *AllElectronics* customers who buy a computer. We can think of each attribute–value pair as an *item*, so the search for these frequent patterns is known as *frequent pattern mining* or *frequent itemset mining*. In Chapters 6 and 7, we saw how **association rules** are derived from frequent patterns, where the associations are commonly used to analyze the purchasing patterns of customers in a store. Such analysis is useful in many decision-making processes such as product placement, catalog design, and cross-marketing.

In this section, we examine how frequent patterns can be used for classification. Section 9.4.1 explores **associative classification**, where association rules are generated from frequent patterns and used for classification. The general idea is that we can search for strong associations between frequent patterns (conjunctions of attribute–value

pairs) and class labels. Section 9.4.2 explores **discriminative frequent pattern–based classification**, where frequent patterns serve as combined features, which are considered in addition to single features when building a classification model. Because frequent patterns explore highly confident associations among multiple attributes, frequent pattern–based classification may overcome some constraints introduced by decision tree induction, which considers only one attribute at a time. Studies have shown many frequent pattern–based classification methods to have greater accuracy and scalability than some traditional classification methods such as C4.5.

9.4.1 Associative Classification

In this section, you will learn about associative classification. The methods discussed are CBA, CMAR, and CPAR.

Before we begin, however, let's look at association rule mining in general. Association rules are mined in a two-step process consisting of *frequent itemset mining* followed by *rule generation*. The first step searches for patterns of attribute–value pairs that occur repeatedly in a data set, where each attribute–value pair is considered an *item*. The resulting attribute–value pairs form *frequent itemsets* (also referred to as *frequent patterns*). The second step analyzes the frequent itemsets to generate association rules. All association rules must satisfy certain criteria regarding their "accuracy" (or *confidence*) and the proportion of the data set that they actually represent (referred to as *support*). For example, the following is an association rule mined from a data set, D, shown with its confidence and support:

$$\begin{aligned} age = youth \wedge credit = OK &\Rightarrow buys\_computer \\ &= yes\ [support = 20\%, confidence = 93\%], \end{aligned} \tag{9.21}$$

where $\wedge$ represents a logical "AND." We will say more about confidence and support later.

More formally, let D be a data set of tuples. Each tuple in D is described by n attributes, $A_1, A_2, \ldots, A_n$, and a class label attribute, A_{class}. All continuous attributes are discretized and treated as categorical (or nominal) attributes. An **item**, p, is an attribute–value pair of the form (A_i, v), where A_i is an attribute taking a value, v. A data tuple $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$ satisfies an item, $p = (A_i, v)$, if and only if $x_i = v$, where x_i is the value of the ith attribute of $\boldsymbol{X}$. Association rules can have any number of items in the rule antecedent (left side) and any number of items in the rule consequent (right side). However, when mining association rules for use in classification, we are only interested in association rules of the form $p_1 \wedge p_2 \wedge \ldots p_l \Rightarrow A_{class} = C$, where the rule antecedent is a conjunction of items, $p_1, p_2, \ldots, p_l$ $(l \leq n)$, associated with a class label, C. For a given rule, R, the percentage of tuples in D satisfying the rule antecedent that also have the class label C is called the **confidence** of R.

From a classification point of view, this is akin to rule accuracy. For example, a confidence of 93% for Rule (9.21) means that 93% of the customers in D who are young and have an OK credit rating belong to the class *buys_computer* = *yes*. The percentage of

tuples in D satisfying the rule antecedent and having class label C is called the **support** of R. A support of 20% for Rule (9.21) means that 20% of the customers in D are young, have an OK credit rating, and belong to the class *buys_computer* = *yes*.

In general, associative classification consists of the following steps:

1. Mine the data for frequent itemsets, that is, find commonly occurring attribute–value pairs in the data.
2. Analyze the frequent itemsets to generate association rules per class, which satisfy confidence and support criteria.
3. Organize the rules to form a rule-based classifier.

Methods of associative classification differ primarily in the approach used for frequent itemset mining and in how the derived rules are analyzed and used for classification. We now look at some of the various methods for associative classification.

One of the earliest and simplest algorithms for associative classification is **CBA** (Classification Based on Associations). CBA uses an iterative approach to frequent itemset mining, similar to that described for Apriori in Section 6.2.1, where multiple passes are made over the data and the derived frequent itemsets are used to generate and test longer itemsets. In general, the number of passes made is equal to the length of the longest rule found. The complete set of rules satisfying minimum confidence and minimum support thresholds are found and then analyzed for inclusion in the classifier. CBA uses a heuristic method to construct the classifier, where the rules are ordered according to decreasing precedence based on their confidence and support. If a set of rules has the same antecedent, then the rule with the highest confidence is selected to represent the set. When classifying a new tuple, the first rule satisfying the tuple is used to classify it. The classifier also contains a default rule, having lowest precedence, which specifies a default class for any new tuple that is not satisfied by any other rule in the classifier. In this way, the set of rules making up the classifier form a *decision list*. In general, CBA was empirically found to be more accurate than C4.5 on a good number of data sets.

CMAR (Classification based on Multiple Association Rules) differs from CBA in its strategy for frequent itemset mining and its construction of the classifier. It also employs several rule pruning strategies with the help of a tree structure for efficient storage and retrieval of rules. CMAR adopts a variant of the *FP-growth* algorithm to find the complete set of rules satisfying the minimum confidence and minimum support thresholds. FP-growth was described in Section 6.2.4. FP-growth uses a tree structure, called an *FP-tree*, to register all the frequent itemset information contained in the given data set, D. This requires only two scans of D. The frequent itemsets are then mined from the FP-tree. CMAR uses an enhanced FP-tree that maintains the distribution of class labels among tuples satisfying each frequent itemset. In this way, it is able to combine rule generation together with frequent itemset mining in a single step.

CMAR employs another tree structure to store and retrieve rules efficiently and to prune rules based on confidence, correlation, and database coverage. Rule pruning strategies are triggered whenever a rule is inserted into the tree. For example, given

two rules, $R1$ and $R2$, if the antecedent of $R1$ is more general than that of $R2$ and $conf(R1) \geq conf(R2)$, then $R2$ is pruned. The rationale is that highly specialized rules with low confidence can be pruned if a more generalized version with higher confidence exists. CMAR also prunes rules for which the rule antecedent and class are not positively correlated, based on an χ^2 test of statistical significance.

"*If more than one rule applies, which one do we use?*" As a classifier, CMAR operates differently than CBA. Suppose that we are given a tuple $\boldsymbol{X}$ to classify and that only one rule satisfies or matches $\boldsymbol{X}$.[4] This case is trivial—we simply assign the rule's class label. Suppose, instead, that more than one rule satisfies $\boldsymbol{X}$. These rules form a set, S. Which rule would we use to determine the class label of $\boldsymbol{X}$? CBA would assign the class label of the most confident rule among the rule set, S. CMAR instead considers multiple rules when making its class prediction. It divides the rules into groups according to class labels. All rules within a group share the same class label and each group has a distinct class label.

CMAR uses a weighted χ^2 measure to find the "strongest" group of rules, based on the statistical correlation of rules within a group. It then assigns $\boldsymbol{X}$ the class label of the strongest group. In this way it considers multiple rules, rather than a single rule with highest confidence, when predicting the class label of a new tuple. In experiments, CMAR had slightly higher average accuracy in comparison with CBA. Its runtime, scalability, and use of memory were found to be more efficient.

"*Is there a way to cut down on the number of rules generated?*" CBA and CMAR adopt methods of frequent itemset mining to generate *candidate* association rules, which include all conjunctions of attribute–value pairs (items) satisfying minimum support. These rules are then examined, and a subset is chosen to represent the classifier. However, such methods generate quite a large number of rules. CPAR (Classification based on Predictive Association Rules) takes a different approach to rule generation, based on a rule generation algorithm for classification known as FOIL (Section 8.4.3). FOIL builds rules to distinguish positive tuples (e.g., *buys_computer* = *yes*) from negative tuples (e.g., *buys_computer* = *no*). For multiclass problems, FOIL is applied to each class. That is, for a class, C, all tuples of class C are considered positive tuples, while the rest are considered negative tuples. Rules are generated to distinguish C tuples from all others. Each time a rule is generated, the positive samples it satisfies (or *covers*) are removed until all the positive tuples in the data set are covered. In this way, fewer rules are generated. CPAR relaxes this step by allowing the covered tuples to remain under consideration, but reducing their weight. The process is repeated for each class. The resulting rules are merged to form the classifier rule set.

During classification, CPAR employs a somewhat different multiple rule strategy than CMAR. If more than one rule satisfies a new tuple, $\boldsymbol{X}$, the rules are divided into groups according to class, similar to CMAR. However, CPAR uses the best k rules of each group to predict the class label of $\boldsymbol{X}$, based on expected accuracy. By considering the best k rules rather than all of a group's rules, it avoids the influence of lower-ranked

[4] If a rule's antecedent satisfies or matches $\boldsymbol{X}$, then we say that the rule satisfies $\boldsymbol{X}$.

rules. CPAR's accuracy on numerous data sets was shown to be close to that of CMAR. However, since CPAR generates far fewer rules than CMAR, it shows much better efficiency with large sets of training data.

In summary, associative classification offers an alternative classification scheme by building rules based on conjunctions of attribute–value pairs that occur frequently in data.

9.4.2 Discriminative Frequent Pattern–Based Classification

From work on associative classification, we see that frequent patterns reflect strong associations between attribute–value pairs (or items) in data and are useful for classification.

"*But just how discriminative are frequent patterns for classification?*" Frequent patterns represent feature combinations. Let's compare the discriminative power of frequent patterns and single features. Figure 9.11 plots the information gain of frequent patterns and single features (i.e., of pattern length 1) for three UCI data sets.[5] The discrimination power of some frequent patterns is higher than that of single features. Frequent patterns map data to a higher dimensional space. They capture more underlying semantics of the data, and thus can hold greater expressive power than single features.

"*Why not consider frequent patterns as combined features, in addition to single features when building a classification model?*" This notion is the basis of **frequent pattern–based classification**—the learning of a classification model in the feature space of single attributes *as well as* frequent patterns. In this way, we transfer the original feature space to a larger space. This will likely increase the chance of including important features.

Let's get back to our earlier question: How discriminative are frequent patterns? Many of the frequent patterns generated in frequent itemset mining are indiscriminative because they are based solely on support, without considering predictive power. That is, by definition, a pattern must satisfy a user-specified minimum support threshold, *min_sup*, to be considered frequent. For example, if *min_sup*, is, say, 5%, a pattern is frequent if it occurs in 5% of the data tuples. Consider Figure 9.12, which plots information gain versus pattern frequency (support) for three UCI data sets. A theoretical upper bound on information gain, which was derived analytically, is also plotted. The figure shows that the discriminative power (assessed here as information gain) of low-frequency patterns is bounded by a small value. This is due to the patterns' limited coverage of the data set. Similarly, the discriminative power of very high-frequency patterns is also bounded by a small value, which is due to their commonness in the data. The upper bound of information gain is a function of pattern frequency. The information gain upper bound increases monotonically with pattern frequency. These observations can be confirmed analytically. Patterns with medium-large supports (e.g., $support = 300$ in Figure 9.12a) may be discriminative or not. Thus, not every frequent pattern is useful.

[5]The University of California at Irvine (UCI) archives several large data sets at *http://kdd.ics.uci.edu/*. These are commonly used by researchers for the testing and comparison of machine learning and data mining algorithms.

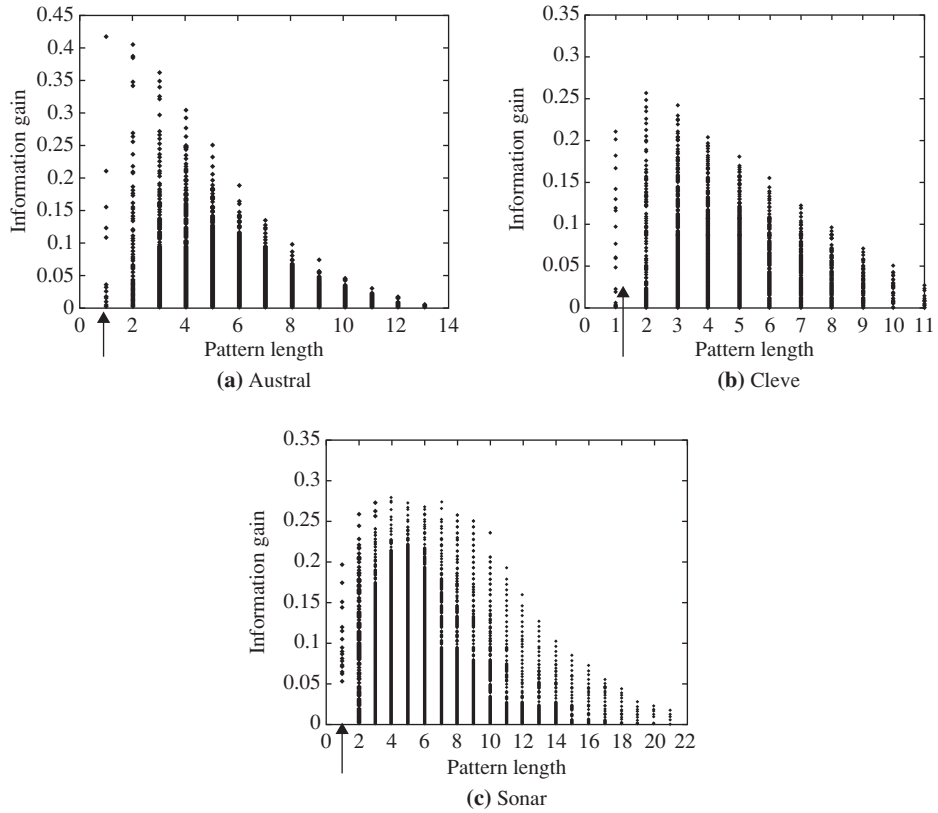


Figure 9.11 Single feature versus frequent pattern: Information gain is plotted for single features (patterns of length 1, indicated by arrows) and frequent patterns (combined features) for three UCI data sets. *Source:* Adapted from Cheng, Yan, Han, and Hsu [CYHH07].

If we were to add all the frequent patterns to the feature space, the resulting feature space would be huge. This slows down the model learning process and may also lead to decreased accuracy due to a form of overfitting in which there are too many features. Many of the patterns may be redundant. Therefore, it's a good idea to apply feature selection to eliminate the less discriminative and redundant frequent patterns as features. The *general framework for discriminative frequent pattern–based classification* is as follows.

1. **Feature generation:** The data, D, are partitioned according to class label. Use frequent itemset mining to discover frequent patterns in each partition, satisfying minimum support. The collection of frequent patterns, $\mathcal{F}$, makes up the feature candidates.
2. **Feature selection:** Apply feature selection to $\mathcal{F}$, resulting in $\mathcal{F}_S$, the set of selected (more discriminating) frequent patterns. Information gain, Fisher score, or other evaluation measures can be used for this step. Relevancy checking can also be

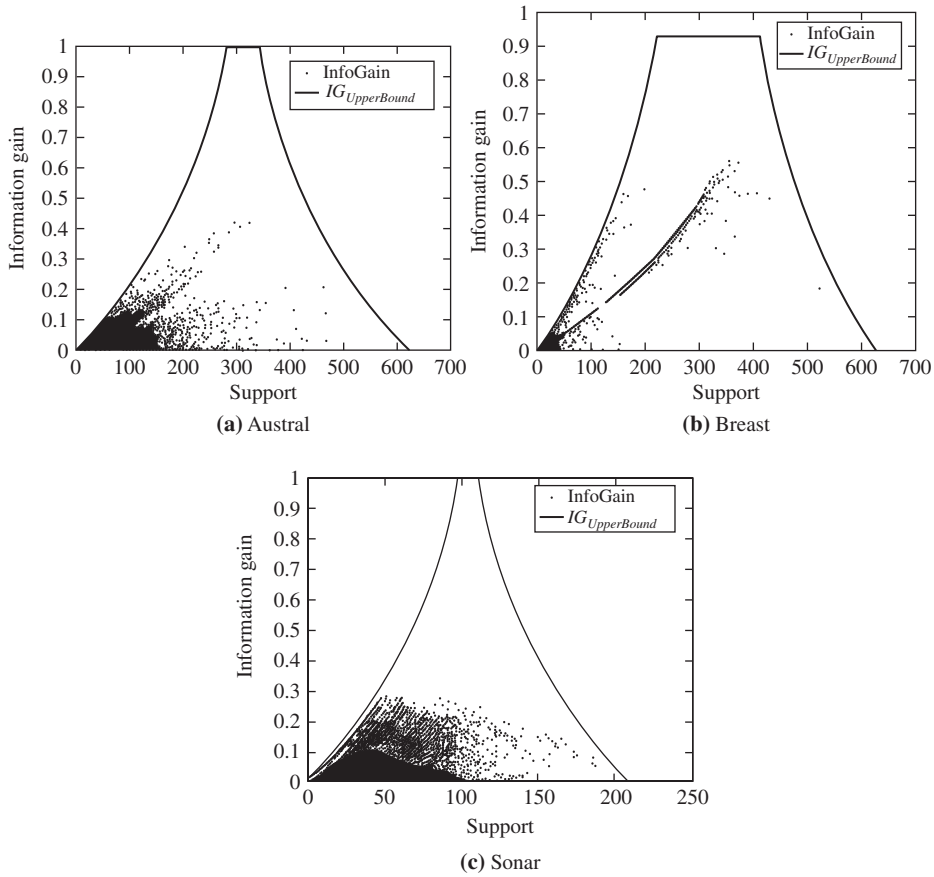


Figure 9.12 Information gain versus pattern frequency (support) for three UCI data sets. A theoretical upper bound on information gain ($IG_{UpperBound}$) is also shown. *Source:* Adapted from Cheng, Yan, Han, and Hsu [CYHH07].

incorporated into this step to weed out redundant patterns. The data set D is transformed to D', where the feature space now includes the single features as well as the selected frequent patterns, $\mathcal{F}_S$.

3. **Learning of classification model:** A classifier is built on the data set D'. Any learning algorithm can be used as the classification model.

The general framework is summarized in Figure 9.13(a), where the discriminative patterns are represented by dark circles. Although the approach is straightforward, we can encounter a computational bottleneck by having to first find *all* the frequent patterns, and then analyze *each one* for selection. The amount of frequent patterns found can be huge due to the explosive number of pattern combinations between items.

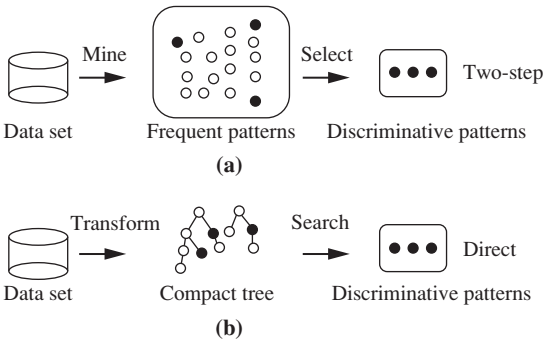


Figure 9.13 A framework for frequent pattern–based classification: (a) a two-step general approach versus (b) the direct approach of DDPMine.

To improve the efficiency of the general framework, consider condensing steps 1 and 2 into just one step. That is, rather than generating the complete set of frequent patterns, it's possible to mine only the highly discriminative ones. This more direct approach is referred to as *direct discriminative pattern mining*. The DDPMine algorithm follows this approach, as illustrated in Figure 9.13(b). It first transforms the training data into a compact tree structure known as a frequent pattern tree, or FP-tree (Section 6.2.4), which holds all of the attribute–value (itemset) association information. It then searches for discriminative patterns on the tree. The approach is direct in that it avoids generating a large number of indiscriminative patterns. It incrementally reduces the problem by eliminating training tuples, thereby progressively shrinking the FP-tree. This further speeds up the mining process.

By choosing to transform the original data to an FP-tree, DDPMine avoids generating redundant patterns because an FP-tree stores only the *closed* frequent patterns. By definition, any subpattern, β, of a closed pattern, α, is redundant with respect to α (Section 6.1.2). DDPMine directly mines the discriminative patterns and integrates feature selection into the mining framework. The theoretical upper bound on information gain is used to facilitate a branch-and-bound search, which prunes the search space significantly. Experimental results show that DDPMine achieves orders of magnitude speedup over the two-step approach without decline in classification accuracy. DDPMine also outperforms state-of-the-art associative classification methods in terms of both accuracy and efficiency.

9.5 Lazy Learners (or Learning from Your Neighbors)

The classification methods discussed so far in this book—decision tree induction, Bayesian classification, rule-based classification, classification by backpropagation, support vector machines, and classification based on association rule mining—are all

examples of *eager learners*. **Eager learners**, when given a set of training tuples, will construct a generalization (i.e., classification) model before receiving new (e.g., test) tuples to classify. We can think of the learned model as being ready and eager to classify previously unseen tuples.

Imagine a contrasting lazy approach, in which the learner instead waits until the last minute before doing any model construction to classify a given test tuple. That is, when given a training tuple, a **lazy learner** simply stores it (or does only a little minor processing) and waits until it is given a test tuple. Only when it sees the test tuple does it perform generalization to classify the tuple based on its similarity to the stored training tuples. Unlike eager learning methods, lazy learners do less work when a training tuple is presented and more work when making a classification or numeric prediction. Because lazy learners store the training tuples or "instances," they are also referred to as **instance-based learners**, even though all learning is essentially based on instances.

When making a classification or numeric prediction, lazy learners can be computationally expensive. They require efficient storage techniques and are well suited to implementation on parallel hardware. They offer little explanation or insight into the data's structure. Lazy learners, however, naturally support incremental learning. They are able to model complex decision spaces having hyperpolygonal shapes that may not be as easily describable by other learning algorithms (such as hyperrectangular shapes modeled by decision trees). In this section, we look at two examples of lazy learners: *k-nearest-neighbor classifiers* (Section 9.5.1) and *case-based reasoning classifiers* (Section 9.5.2).

9.5.1 *k*-Nearest-Neighbor Classifiers

The *k*-nearest-neighbor method was first described in the early 1950s. The method is labor intensive when given large training sets, and did not gain popularity until the 1960s when increased computing power became available. It has since been widely used in the area of pattern recognition.

Nearest-neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by n attributes. Each tuple represents a point in an n-dimensional space. In this way, all the training tuples are stored in an n-dimensional pattern space. When given an unknown tuple, a ***k*-nearest-neighbor classifier** searches the pattern space for the k training tuples that are closest to the unknown tuple. These k training tuples are the k "nearest neighbors" of the unknown tuple.

"Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $\boldsymbol{X}_1 = (x_{11}, x_{12}, \ldots, x_{1n})$ and $\boldsymbol{X}_2 = (x_{21}, x_{22}, \ldots, x_{2n})$, is

$$dist(\boldsymbol{X}_1, \boldsymbol{X}_2) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}. \tag{9.22}$$

In other words, for each numeric attribute, we take the difference between the corresponding values of that attribute in tuple $\boldsymbol{X}_1$ and in tuple $\boldsymbol{X}_2$, square this difference, and accumulate it. The square root is taken of the total accumulated distance count. Typically, we normalize the values of each attribute before using Eq. (9.22). This helps prevent attributes with initially large ranges (e.g., *income*) from outweighing attributes with initially smaller ranges (e.g., binary attributes). Min-max normalization, for example, can be used to transform a value v of a numeric attribute A to v' in the range $[0, 1]$ by computing

$$v' = \frac{v - min_A}{max_A - min_A}, \tag{9.23}$$

where min_A and max_A are the minimum and maximum values of attribute A. Chapter 3 describes other methods for data normalization as a form of data transformation.

For k-nearest-neighbor classification, the unknown tuple is assigned the most common class among its k-nearest neighbors. When $k = 1$, the unknown tuple is assigned the class of the training tuple that is closest to it in pattern space. Nearest-neighbor classifiers can also be used for numeric prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the average value of the real-valued labels associated with the k-nearest neighbors of the unknown tuple.

"*But how can distance be computed for attributes that are not numeric, but nominal (or categorical) such as color?*" The previous discussion assumes that the attributes used to describe the tuples are all numeric. For nominal attributes, a simple method is to compare the corresponding value of the attribute in tuple $\boldsymbol{X}_1$ with that in tuple $\boldsymbol{X}_2$. If the two are identical (e.g., tuples $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ both have the color blue), then the difference between the two is taken as 0. If the two are different (e.g., tuple $\boldsymbol{X}_1$ is blue but tuple $\boldsymbol{X}_2$ is red), then the difference is considered to be 1. Other methods may incorporate more sophisticated schemes for differential grading (e.g., where a larger difference score is assigned, say, for blue and white than for blue and black).

"*What about missing values?*" In general, if the value of a given attribute A is missing in tuple $\boldsymbol{X}_1$ and/or in tuple $\boldsymbol{X}_2$, we assume the maximum possible difference. Suppose that each of the attributes has been mapped to the range $[0, 1]$. For nominal attributes, we take the difference value to be 1 if either one or both of the corresponding values of A are missing. If A is numeric and missing from both tuples $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$, then the difference is also taken to be 1. If only one value is missing and the other (which we will call v') is present and normalized, then we can take the difference to be either $|1 - v'|$ or $|0 - v'|$ (i.e., $1 - v'$ or v'), whichever is greater.

"*How can I determine a good value for k, the number of neighbors?*" This can be determined experimentally. Starting with $k = 1$, we use a test set to estimate the error rate of the classifier. This process can be repeated each time by incrementing k to allow for one more neighbor. The k value that gives the minimum error rate may be selected. In general, the larger the number of training tuples, the larger the value of k will be (so that classification and numeric prediction decisions can be based on a larger portion of the stored tuples). As the number of training tuples approaches infinity and $k = 1$, the

error rate can be no worse than twice the Bayes error rate (the latter being the theoretical minimum). If k also approaches infinity, the error rate approaches the Bayes error rate.

Nearest-neighbor classifiers use distance-based comparisons that intrinsically assign equal weight to each attribute. They therefore can suffer from poor accuracy when given noisy or irrelevant attributes. The method, however, has been modified to incorporate attribute weighting and the pruning of noisy data tuples. The choice of a distance metric can be critical. The Manhattan (city block) distance (Section 2.4.4), or other distance measurements, may also be used.

Nearest-neighbor classifiers can be extremely slow when classifying test tuples. If D is a training database of $|D|$ tuples and $k = 1$, then $O(|D|)$ comparisons are required to classify a given test tuple. By presorting and arranging the stored tuples into search trees, the number of comparisons can be reduced to $O(log(|D|)$. Parallel implementation can reduce the running time to a constant, that is, $O(1)$, which is independent of $|D|$.

Other techniques to speed up classification time include the use of *partial distance* calculations and *editing* the stored tuples. In the **partial distance** method, we compute the distance based on a subset of the n attributes. If this distance exceeds a threshold, then further computation for the given stored tuple is halted, and the process moves on to the next stored tuple. The **editing** method removes training tuples that prove useless. This method is also referred to as **pruning** or **condensing** because it reduces the total number of tuples stored.

9.5.2 Case-Based Reasoning

Case-based reasoning (CBR) classifiers use a database of problem solutions to solve new problems. Unlike nearest-neighbor classifiers, which store training tuples as points in Euclidean space, CBR stores the tuples or "cases" for problem solving as complex symbolic descriptions. Business applications of CBR include problem resolution for customer service help desks, where cases describe product-related diagnostic problems. CBR has also been applied to areas such as engineering and law, where cases are either technical designs or legal rulings, respectively. Medical education is another area for CBR, where patient case histories and treatments are used to help diagnose and treat new patients.

When given a new case to classify, a case-based reasoner will first check if an identical training case exists. If one is found, then the accompanying solution to that case is returned. If no identical case is found, then the case-based reasoner will search for training cases having components that are similar to those of the new case. Conceptually, these training cases may be considered as neighbors of the new case. If cases are represented as graphs, this involves searching for subgraphs that are similar to subgraphs within the new case. The case-based reasoner tries to combine the solutions of the neighboring training cases to propose a solution for the new case. If incompatibilities arise with the individual solutions, then backtracking to search for other solutions may be necessary. The case-based reasoner may employ background knowledge and problem-solving strategies to propose a feasible combined solution.

Challenges in case-based reasoning include finding a good similarity metric (e.g., for matching subgraphs) and suitable methods for combining solutions. Other challenges include the selection of salient features for indexing training cases and the development of efficient indexing techniques. A trade-off between accuracy and efficiency evolves as the number of stored cases becomes very large. As this number increases, the case-based reasoner becomes more intelligent. After a certain point, however, the system's efficiency will suffer as the time required to search for and process relevant cases increases. As with nearest-neighbor classifiers, one solution is to edit the training database. Cases that are redundant or that have not proved useful may be discarded for the sake of improved performance. These decisions, however, are not clear-cut and their automation remains an active area of research.

9.6 Other Classification Methods

In this section, we give a brief description of several other classification methods, including genetic algorithms (Section 9.6.1), rough set approach (Section 9.6.2), and fuzzy set approaches (Section 9.6.3). In general, these methods are less commonly used for classification in commercial data mining systems than the methods described earlier in this book. However, these methods show their strength in certain applications, and hence it is worthwhile to include them here.

9.6.1 Genetic Algorithms

Genetic algorithms attempt to incorporate ideas of natural evolution. In general, genetic learning starts as follows. An initial **population** is created consisting of randomly generated rules. Each rule can be represented by a string of bits. As a simple example, suppose that samples in a given training set are described by two Boolean attributes, A_1 and A_2, and that there are two classes, C_1 and C_2. The rule "*IF* A_1 *AND NOT* A_2 *THEN* C_2" can be encoded as the bit string "100," where the two leftmost bits represent attributes A_1 and A_2, respectively, and the rightmost bit represents the class. Similarly, the rule "*IF NOT* A_1 *AND NOT* A_2 *THEN* C_1" can be encoded as "001." If an attribute has k values, where $k > 2$, then k bits may be used to encode the attribute's values. Classes can be encoded in a similar fashion.

Based on the notion of survival of the fittest, a new population is formed to consist of the *fittest* rules in the current population, as well as *offspring* of these rules. Typically, the **fitness** of a rule is assessed by its classification accuracy on a set of training samples.

Offspring are created by applying genetic operators such as crossover and mutation. In **crossover**, substrings from pairs of rules are swapped to form new pairs of rules. In **mutation**, randomly selected bits in a rule's string are inverted.

The process of generating new populations based on prior populations of rules continues until a population, P, evolves where each rule in P satisfies a prespecified fitness threshold.

Genetic algorithms are easily parallelizable and have been used for classification as well as other optimization problems. In data mining, they may be used to evaluate the fitness of other algorithms.

9.6.2 Rough Set Approach

Rough set theory can be used for classification to discover structural relationships within imprecise or noisy data. It applies to discrete-valued attributes. Continuous-valued attributes must therefore be discretized before its use.

Rough set theory is based on the establishment of **equivalence classes** within the given training data. All the data tuples forming an equivalence class are indiscernible, that is, the samples are identical with respect to the attributes describing the data. Given real-world data, it is common that some classes cannot be distinguished in terms of the available attributes. Rough sets can be used to approximately or "roughly" define such classes. A rough set definition for a given class, C, is approximated by two sets—a **lower approximation** of C and an **upper approximation** of C. The lower approximation of C consists of all the data tuples that, based on the knowledge of the attributes, are certain to belong to C without ambiguity. The upper approximation of C consists of all the tuples that, based on the knowledge of the attributes, cannot be described as not belonging to C. The lower and upper approximations for a class C are shown in Figure 9.14, where each rectangular region represents an equivalence class. Decision rules can be generated for each class. Typically, a decision table is used to represent the rules.

Rough sets can also be used for attribute subset selection (or feature reduction, where attributes that do not contribute to the classification of the given training data can be identified and removed) and relevance analysis (where the contribution or significance of each attribute is assessed with respect to the classification task). The problem of finding the minimal subsets (**reducts**) of attributes that can describe all the concepts in the given data set is NP-hard. However, algorithms to reduce the computation intensity have been proposed. In one method, for example, a **discernibility matrix** is used that stores the differences between attribute values for each pair of data tuples. Rather than

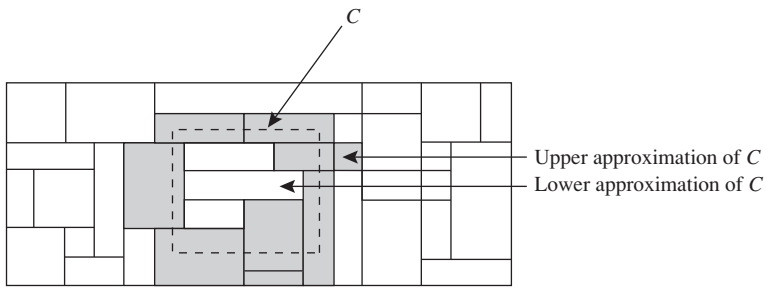


Figure 9.14 A rough set approximation of class C's set of tuples using lower and upper approximation sets of C. The rectangular regions represent equivalence classes.

searching on the entire training set, the matrix is instead searched to detect redundant attributes.

9.6.3 Fuzzy Set Approaches

Rule-based systems for classification have the disadvantage that they involve sharp cutoffs for continuous attributes. For example, consider the following rule for customer credit application approval. The rule essentially says that applications for customers who have had a job for two or more years and who have a high income (i.e., of at least $50,000) are approved:

$$IF\ (years\_employed \geq 2)\ AND\ (income \geq 50{,}000)\ THEN\ credit = approved. \quad (9.24)$$

By Rule (9.24), a customer who has had a job for at least two years will receive credit if her income is, say, $50,000, but not if it is $49,000. Such harsh thresholding may seem unfair.

Instead, we can discretize *income* into categories (e.g., {*low_income, medium_income, high_income*}) and then apply **fuzzy logic** to allow "fuzzy" thresholds or boundaries to be defined for each category (Figure 9.15). Rather than having a precise cutoff between categories, fuzzy logic uses truth values between 0.0 and 1.0 to represent the degree of membership that a certain value has in a given category. Each category then represents a **fuzzy set**. Hence, with fuzzy logic, we can capture the notion that an income of $49,000 is, more or less, high, although not as high as an income of $50,000. Fuzzy logic systems typically provide graphical tools to assist users in converting attribute values to fuzzy truth values.

Fuzzy set theory is also known as **possibility theory**. It was proposed by Lotfi Zadeh in 1965 as an alternative to traditional two-value logic and probability theory. It lets us work at a high abstraction level and offers a means for dealing with imprecise data

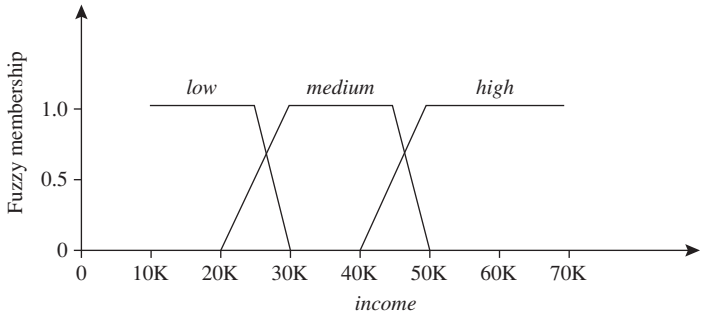


Figure 9.15 Fuzzy truth values for *income*, representing the degree of membership of *income* values with respect to the categories {*low, medium, high*}. Each category represents a fuzzy set. Note that a given income value, x, can have membership in more than one fuzzy set. The membership values of x in each fuzzy set do not have to total to 1.

measurement. Most important, fuzzy set theory allows us to deal with vague or inexact facts. For example, being a member of a set of high incomes is inexact (e.g., if $50,000 is high, then what about $49,000? or $48,000?) Unlike the notion of traditional "crisp" sets where an element belongs to either a set S or its complement, in fuzzy set theory, elements can belong to more than one fuzzy set. For example, the income value $49,000 belongs to both the *medium* and *high* fuzzy sets, but to differing degrees. Using fuzzy set notation and following Figure 9.15, this can be shown as

$$m_{medium\_income}(\$49{,}000) = 0.15 \; and \; m_{high\_income}(\$49{,}000) = 0.96,$$

where m denotes the membership function, that is operating on the fuzzy sets of *medium_income* and *high_income*, respectively. In fuzzy set theory, membership values for a given element, x (e.g., for $49,000), do not have to sum to 1. This is unlike traditional probability theory, which is constrained by a summation axiom.

Fuzzy set theory is useful for data mining systems performing rule-based classification. It provides operations for combining fuzzy measurements. Suppose that in addition to the fuzzy sets for *income*, we defined the fuzzy sets *junior_employee* and *senior_employee* for the attribute *years_employed*. Suppose also that we have a rule that, say, tests *high_income* and *senior_employee* in the rule antecedent (IF part) for a given employee, x. If these two fuzzy measures are ANDed together, the minimum of their measure is taken as the measure of the rule. In other words,

$$m_{(high\_income \; AND \; senior\_employee)}(x) = min(m_{high\_income}(x), m_{senior\_employee}(x)).$$

This is akin to saying that a chain is as strong as its weakest link. If the two measures are ORed, the maximum of their measure is taken as the measure of the rule. In other words,

$$m_{(high\_income \; OR \; senior\_employee)}(x) = max(m_{high\_income}(x), m_{senior\_employee}(x)).$$

Intuitively, this is like saying that a rope is as strong as its strongest strand.

Given a tuple to classify, more than one fuzzy rule may apply. Each applicable rule contributes a vote for membership in the categories. Typically, the truth values for each predicted category are summed, and these sums are combined. Several procedures exist for translating the resulting fuzzy output into a *defuzzified* or crisp value that is returned by the system.

Fuzzy logic systems have been used in numerous areas for classification, including market research, finance, health care, and environmental engineering.

9.7 Additional Topics Regarding Classification

Most of the classification algorithms we have studied handle multiple classes, but some, such as support vector machines, assume only two classes exist in the data. What adaptations can be made to allow for when there are more than two classes? This question is addressed in Section 9.7.1 on *multiclass classification*.

What can we do if we want to build a classifier for data where only some of the data are class-labeled, but most are not? Document classification, speech recognition, and information extraction are just a few examples of applications in which unlabeled data are abundant. Consider *document classification*, for example. Suppose we want to build a model to automatically classify text documents like articles or web pages. In particular, we want the model to distinguish between hockey and football documents. We have a vast amount of documents available, yet the documents are not class-labeled. Recall that supervised learning requires a training set, that is, a set of classlabeled data. To have a human examine and assign a class label to individual documents (to form a training set) is time consuming and expensive.

Speech recognition requires the accurate labeling of speech utterances by trained linguists. It was reported that 1 minute of speech takes 10 minutes to label, and annotating phonemes (basic units of sound) can take 400 times as long. *Information extraction systems* are trained using labeled documents with detailed annotations. These are obtained by having human experts highlight items or relations of interest in text such as the names of companies or individuals. High-level expertise may be required for certain knowledge domains such as gene and disease mentions in biomedical information extraction. Clearly, the manual assignment of class labels to prepare a training set can be extremely costly, time consuming, and tedious.

We study three approaches to classification that are suitable for situations where there is an abundance of unlabeled data. Section 9.7.2 introduces semisupervised classification, which builds a classifier using both labeled and unlabeled data. Section 9.7.3 presents *active learning*, where the learning algorithm carefully selects a few of the unlabeled data tuples and asks a human to label only those tuples. Section 9.7.4 presents *transfer learning*, which aims to extract the knowledge from one or more source tasks (e.g., classifying camera reviews) and apply the knowledge to a target task (e.g., TV reviews). Each of these strategies can reduce the need to annotate large amounts of data, resulting in cost and time savings.

9.7.1 Multiclass Classification

Some classification algorithms, such as support vector machines, are designed for binary classification. How can we extend these algorithms to allow for **multiclass classification** (i.e., classification involving more than two classes)?

A simple approach is **one-versus-all** (OVA). Given m classes, we train m binary classifiers, one for each class. Classifier j is trained using tuples of class j as the positive class, and the remaining tuples as the negative class. It learns to return a positive value for class j and a negative value for the rest. To classify an unknown tuple, $\boldsymbol{X}$, the set of classifiers vote as an ensemble. For example, if classifier j predicts the positive class for $\boldsymbol{X}$, then class j gets one vote. If it predicts the negative class for $\boldsymbol{X}$, then each of the classes except j gets one vote. The class with the most votes is assigned to $\boldsymbol{X}$.

All-versus-all (AVA) is an alternative approach that learns a classifier for each pair of classes. Given m classes, we construct $\frac{m(m-1)}{2}$ binary classifiers. A classifier is trained

using tuples of the two classes it should discriminate. To classify an unknown tuple, each classifier votes. The tuple is assigned the class with the maximum number of votes. All-versus-all tends to be superior to one-versus-all.

A problem with the previous schemes is that binary classifiers are sensitive to errors. If any classifier makes an error, it can affect the vote count.

Error-correcting codes can be used to improve the accuracy of multiclass classification, not just in the previous situations, but for classification in general. Error-correcting codes were originally designed to correct errors during data transmission for communication tasks. For such tasks, the codes are used to add redundancy to the data being transmitted so that, even if some errors occur due to noise in the channel, the data can be correctly received at the other end. For multiclass classification, even if some of the individual binary classifiers make a prediction error for a given unknown tuple, we may still be able to correctly label the tuple.

An error-correcting code is assigned to each class, where each code is a bit vector. Figure 9.16 show an example of 7-bit codewords assigned to classes C_1, C_2, C_3, and C_4. We train one classifier for each bit position. Therefore, in our example we train seven classifiers. If a classifier makes an error, there is a better chance that we may still be able to predict the right class for a given unknown tuple because of the redundancy gained by having additional bits. The technique uses a distance measurement called the Hamming distance to guess the "closest" class in case of errors, and is illustrated in Example 9.3.

Example 9.3 **Multiclass classification with error-correcting codes.** Consider the 7-bit codewords associated with classes C_1 to C_4 in Figure 9.16. Suppose that, given an unknown tuple to label, the seven trained binary classifiers collectively output the codeword 0001010, which does not match a codeword for any of the four classes. A classification error has obviously occurred, but can we figure out what the classification most likely should be? We can try by using the **Hamming distance**, which is the number of different bits between two codewords. The Hamming distance between the output codeword and the codeword for C_1 is 5 because five bits—namely, the first, second, third, fifth, and seventh—differ. Similarly, the Hamming distance between the output code and the codewords for C_2 through C_4 are 3, 3, and 1, respectively. Note that the output codeword is closest to the codeword for C_4. That is, the smallest Hamming distance between the output and a class codeword is for class C_4. Therefore, we assign C_4 as the class label of the given tuple. ■

Class	*Error-correcting codeword*
C_1	1 1 1 1 1 1 1
C_2	0 0 0 0 1 1 1
C_3	0 0 1 1 0 0 1
C_4	0 1 0 1 0 1 0

Figure 9.16 Error-correcting codes for a multiclass classification problem involving four classes.

Error-correcting codes can correct up to $\frac{h-1}{2}$ 1-bit errors, where h is the minimum Hamming distance between any two codewords. If we use one bit per class, such as for 4-bit codewords for classes C_1 through C_4, then this is equivalent to the one-versus-all approach, and the codes are not sufficient to self-correct. (Try it as an exercise.) When selecting error-correcting codes for multiclass classification, there must be good row-wise and column-wise separation between the codewords. The greater the distance, the more likely that errors will be corrected.

9.7.2 Semi-Supervised Classification

Semi-supervised classification uses labeled data and unlabeled data to build a classifier. Let $X_l = \{(x_1, y_1), \ldots, x_l, y_l)\}$ be the set of labeled data and $X_u = \{x_{l+1}, \ldots, x_n\}$ be the set of unlabeled data. Here we describe a few examples of this approach for learning.

Self-training is the simplest form of semi-supervised classification. It first builds a classifier using the labeled data. The classifier then tries to label the unlabeled data. The tuple with the most confident label prediction is added to the set of labeled data, and the process repeats (Figure 9.17). Although the method is easy to understand, a disadvantage is that it may reinforce errors.

Cotraining is another form of semi-supervised classification, where two or more classifiers teach each other. Each learner uses a different and ideally independent set of features for each tuple. Consider web page data, for example, where attributes relating to the images on the page may be used as one set of features, while attributes relating to the corresponding text constitute another set of features for the same data. Each set

Self-training

1. Select a learning method such as, say, Bayesian classification. Build the classifier using the labeled data, X_l.
2. Use the classifier to label the unlabeled data, X_u.
3. Select the tuple $x \in X_u$ having the highest confidence (most confident prediction). Add it and its predicted label to X_l.
4. Repeat (i.e., retrain the classifier using the augmented set of labeled data).

Cotraining

1. Define two separate nonoverlapping feature sets for the labeled data, X_l.
2. Train two classifiers, f_1 and f_2, on the labeled data, where f_1 is trained using one of the feature sets and f_2 is trained using the other.
3. Classify X_u with f_1 and f_2 separately.
4. Add the most confident $(x, f_1(x))$ to the set of labeled data used by f_2, where $x \in X_u$. Similarly, add the most confident $(x, f_2(x))$ to the set of labeled data used by f_1.
5. Repeat.

Figure 9.17 Self-training and cotraining methods of semi-supervised classification.

of features should be sufficient to train a good classifier. Suppose we split the feature set into two sets and train two classifiers, f_1 and f_2, where each classifier is trained on a different set. Then, f_1 and f_2 are used to predict the class labels for the unlabeled data, X_u. Each classifier then teaches the other in that the tuple having the most confident prediction from f_1 is added to the set of labeled data for f_2 (along with its label).

Similarly, the tuple having the most confident prediction from f_2 is added to the set of labeled data for f_1. The method is summarized in Figure 9.17. Cotraining is less sensitive to errors than self-training. A difficulty is that the assumptions for its usage may not hold true, that is, it may not be possible to split the features into mutually exclusive and class-conditionally independent sets.

Alternate approaches to semi-supervised learning exist. For example, we can model the joint probability distribution of the features and the labels. For the unlabeled data, the labels can then be treated as missing data. The EM algorithm (Chapter 11) can be used to maximize the likelihood of the model. Methods using support vector machines have also been proposed.

9.7.3 Active Learning

Active learning is an iterative type of supervised learning that is suitable for situations where data are abundant, yet the class labels are scarce or expensive to obtain. The learning algorithm is active in that it can purposefully query a user (e.g., a human oracle) for labels. The number of tuples used to learn a concept this way is often much smaller than the number required in typical supervised learning.

"*How does active learning work to overcome the labeling bottleneck?*" To keep costs down, the active learner aims to achieve high accuracy using as few labeled instances as possible. Let D be all of data under consideration. Various strategies exist for active learning on D. Figure 9.18 illustrates a *pool-based approach* to active learning. Suppose that a small subset of D is class-labeled. This set is denoted L. U is the set of unlabeled data in D. It is also referred to as a pool of unlabeled data. An active learner begins with L as the initial training set. It then uses a *querying function* to carefully select one or more data samples from U and requests labels for them from an oracle (e.g., a human annotator). The newly labeled samples are added to L, which the learner then uses in a standard supervised way. The process repeats. The active learning goal is to achieve high accuracy using as few labeled tuples as possible. Active learning algorithms are typically evaluated with the use of learning curves, which plot accuracy as a function of the number of instances queried.

Most of the active learning research focuses on how to *choose* the data tuples to be queried. Several frameworks have been proposed. *Uncertainty sampling* is the most common, where the active learner chooses to query the tuples which it is the least certain how to label. Other strategies work to reduce the *version space*, that is, the subset of all hypotheses that are consistent with the observed training tuples. Alternatively, we may follow a decision-theoretic approach that estimates expected error reduction. This selects tuples that would result in the greatest reduction in the total number of

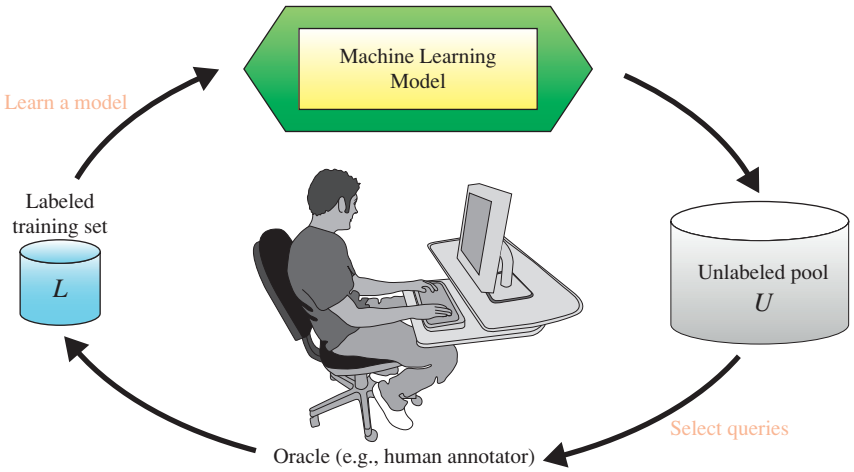


Figure 9.18 The pool-based active learning cycle. *Source:* From Settles [Set10], Burr Settles Computer Sciences Technical Report 1648, University of Wisconsin–Madison; used with permission.

incorrect predictions such as by reducing the expected entropy over U. This latter approach tends to be more computationally expensive.

9.7.4 Transfer Learning

Suppose that *AllElectronics* has collected a number of customer reviews on a product such as a brand of camera. The classification task is to automatically label the reviews as either positive or negative. This task is known as **sentiment classification**. We could examine each review and annotate it by adding a *positive* or *negative* class label. The labeled reviews can then be used to train and test a classifier to label future reviews of the product as either positive or negative. The manual effort involved in annotating the review data can be expensive and time consuming.

Suppose that *AllElectronics* has customer reviews for other products as well such as TVs. The distribution of review data for different types of products can vary greatly. We cannot assume that the TV-review data will have the same distribution as the camera-review data; thus we must build a separate classification model for the TV-review data. Examining and labeling the TV-review data to form a training set will require a lot of effort. In fact, we would need to label a large amount of the data to train the review-classification models for each product. It would be nice if we could adapt an existing classification model (e.g., the one we built for cameras) to help learn a classification model for TVs. Such *knowledge transfer* would reduce the need to annotate a large amount of data, resulting in cost and time savings. This is the essence behind *transfer learning*.

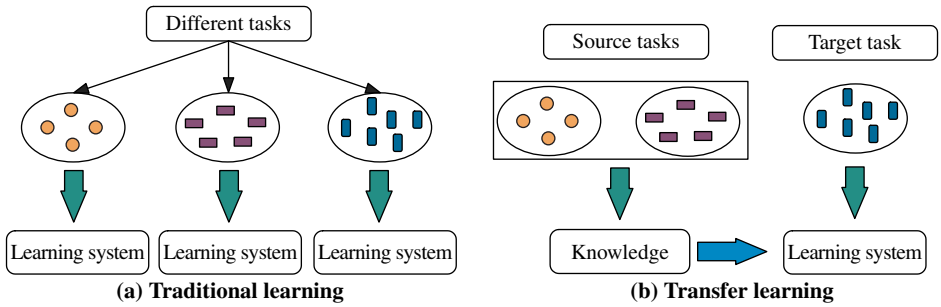


Figure 9.19 Transfer learning versus traditional learning. (a) Traditional learning methods build a new classifier from scratch for each classification task. (b) Transfer learning applies knowledge from a source classifier to simplify the construction of a classifier for a new, target task. *Source:* From Pan and Yang [PY10]; used with permission.

Transfer learning aims to extract the knowledge from one or more *source tasks* and apply the knowledge to a *target task*. In our example, the source task is the classification of camera reviews, and the target task is the classification of TV reviews. Figure 9.19 illustrates a comparison between traditional learning methods and transfer learning. Traditional learning methods build a new classifier for each new classification task, based on available class-labeled training and test data. Transfer learning algorithms apply knowledge about source tasks when building a classifier for a new (target) task. Construction of the resulting classifier requires fewer training data and less training time. Traditional learning algorithms assume that the training data and test data are drawn from the same distribution and the same feature space. Thus, if the distribution changes, such methods need to rebuild the models from scratch.

Transfer learning allows the distributions, tasks, and even the data domains used in training and testing to be different. Transfer learning is analogous to the way humans may apply their knowledge of a task to facilitate the learning of another task. For example, if we know how to play the recorder, we may apply our knowledge of note reading and music to simplify the task of learning to play the piano. Similarly, knowing Spanish may make it easier to learn Italian.

Transfer learning is useful for common applications where the data become outdated or the distribution changes. Here we give two more examples. Consider *web-document classification*, where we may have trained a classifier to label, say, articles from various newsgroups according to predefined categories. The web data that were used to train the classifier can easily become outdated because the topics on the Web change frequently. Another application area for transfer learning is *email spam filtering*. We could train a classifier to label email as either "*spam*" or "*not spam*," using email from a group of users. If new users come along, the distribution of their email can be different from the original group, hence the need to adapt the learned model to incorporate the new data.

There are various approaches to transfer learning, the most common of which is the *instance-based transfer learning* approach. This approach reweights some of the data from the source task and uses it to learn the target task. The **TrAdaBoost** (Transfer AdaBoost) algorithm exemplifies this approach. Consider our previous example of web-document classification, where the distribution of the old data on which the classifier was trained (the source data) is different from the newer data (the target data). TrAdaBoost assumes that the source and target domain data are each described by the same set of attributes (i.e., they have the same "feature space") and the same set of class labels, but that the distribution of the data in the two domains is very different. It extends the AdaBoost ensemble method described in Section 8.6.3. TrAdaBoost requires the labeling of only a small amount of the target data. Rather than throwing out all the old source data, TrAdaBoost assumes that a large amount of it can be useful in training the new classification model. The idea is to filter out the influence of any old data that are very different from the new data by automatically adjusting weights assigned to the training tuples.

Recall that in boosting, an ensemble is created by learning a series of classifiers. To begin, each tuple is assigned a weight. After a classifier M_i is learned, the weights are updated to allow the subsequent classifier, M_{i+1}, to "pay more attention" to the training tuples that were misclassified by M_i. TrAdaBoost follows this strategy for the target data. However, if a source data tuple is misclassified, TrAdaBoost reasons that the tuple is probably very different from the target data. It therefore *reduces* the weight of such tuples so that they will have less effect on the subsequent classifier. As a result, TrAdaBoost can learn an accurate classification model using only a small amount of new data and a large amount of old data, even when the new data alone are insufficient to train the model. Hence, in this way TrAdaBoost allows knowledge to be transferred from the old classifier to the new one.

A challenge with transfer learning is **negative transfer**, which occurs when the new classifier performs worse than if there had been no transfer at all. Work on how to avoid negative transfer is an area of future research. *Heterogeneous transfer learning*, which involves transferring knowledge from different feature spaces and multiple source domains, is another venue for further work. Much of the research on transfer learning to date has been on small-scale applications. The use of transfer learning on larger applications, such as social network analysis and video classification, is an area for further investigation.

9.8 Summary

- Unlike naïve Bayesian classification (which assumes class conditional independence), **Bayesian belief networks** allow class conditional independencies to be defined between subsets of variables. They provide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification.

- **Backpropagation** is a neural network algorithm for classification that employs a method of gradient descent. It searches for a set of weights that can model the data so as to minimize the mean-squared distance between the network's class prediction and the actual class label of data tuples. Rules may be extracted from trained neural networks to help improve the interpretability of the learned network.
- A **support vector machine** is an algorithm for the classification of both linear and nonlinear data. It transforms the original data into a higher dimension, from where it can find a hyperplane for data separation using essential training tuples called **support vectors**.
- *Frequent patterns* reflect strong associations between attribute–value pairs (or items) in data and are used in **classification based on frequent patterns**. Approaches to this methodology include associative classification and discriminant frequent pattern–based classification. In **associative classification**, a classifier is built from association rules generated from frequent patterns. In **discriminative frequent pattern–based classification**, frequent patterns serve as combined features, which are considered in addition to single features when building a classification model.
- Decision tree classifiers, Bayesian classifiers, classification by backpropagation, support vector machines, and classification based on frequent patterns are all examples of **eager learners** in that they use training tuples to construct a generalization model and in this way are ready for classifying new tuples. This contrasts with **lazy learners** or **instance-based** methods of classification, such as nearest-neighbor classifiers and case-based reasoning classifiers, which store all of the training tuples in pattern space and wait until presented with a test tuple before performing generalization. Hence, lazy learners require efficient indexing techniques.
- In **genetic algorithms**, populations of rules "evolve" via operations of crossover and mutation until all rules within a population satisfy a specified threshold. **Rough set theory** can be used to approximately define classes that are not distinguishable based on the available attributes. **Fuzzy set** approaches replace "brittle" threshold cutoffs for continuous-valued attributes with membership degree functions.
- Binary classification schemes, such as support vector machines, can be adapted to handle **multiclass classification**. This involves constructing an ensemble of binary classifiers. Error-correcting codes can be used to increase the accuracy of the ensemble.
- **Semi-supervised classification** is useful when large amounts of unlabeled data exist. It builds a classifier using both labeled and unlabeled data. Examples of semi-supervised classification include *self-training* and *cotraining*.
- **Active learning** is a form of supervised learning that is also suitable for situations where data are abundant, yet the class labels are scarce or expensive to obtain. The learning algorithm can actively query a user (e.g., a human oracle) for labels. To keep costs down, the active learner aims to achieve high accuracy using as few labeled instances as possible.

- **Transfer learning** aims to extract the knowledge from one or more *source tasks* and apply the knowledge to a *target task*. TrAdaBoost is an example of the *instance-based approach* to transfer learning, which reweights some of the data from the source task and uses it to learn the target task, thereby requiring fewer labeled target-task tuples.

9.9 Exercises

9.1 The following table consists of training data from an employee database. The data have been generalized. For example, "31 ... 35" for *age* represents the age range of 31 to 35. For a given row entry, *count* represents the number of data tuples having the values for *department, status, age*, and *salary* given in that row.

department	*status*	*age*	*salary*	*count*
sales	senior	31 ... 35	46K ... 50K	30
sales	junior	26 ... 30	26K ... 30K	40
sales	junior	31 ... 35	31K ... 35K	40
systems	junior	21 ... 25	46K ... 50K	20
systems	senior	31 ... 35	66K ... 70K	5
systems	junior	26 ... 30	46K ... 50K	3
systems	senior	41 ... 45	66K ... 70K	3
marketing	senior	36 ... 40	46K ... 50K	10
marketing	junior	31 ... 35	41K ... 45K	4
secretary	senior	46 ... 50	36K ... 40K	4
secretary	junior	26 ... 30	26K ... 30K	6

Let *status* be the class-label attribute.

(a) Design a multilayer feed-forward neural network for the given data. Label the nodes in the input and output layers.

(b) Using the multilayer feed-forward neural network obtained in (a), show the weight values after one iteration of the backpropagation algorithm, given the training instance "*(sales, senior, 31 ... 35, 46K ... 50K)*". Indicate your initial weight values and biases and the learning rate used.

9.2 The *support vector machine* is a highly accurate classification method. However, SVM classifiers suffer from slow processing when training with a large set of data tuples. Discuss how to overcome this difficulty and develop a scalable SVM algorithm for efficient SVM classification in large data sets.

9.3 Compare and contrast *associative classification* and *discriminative frequent pattern–based classification*. Why is classification based on frequent patterns able to achieve higher classification accuracy in many cases than a classic decision tree method?

9.4 Compare the advantages and disadvantages of *eager* classification (e.g., decision tree, Bayesian, neural network) versus *lazy* classification (e.g., *k*-nearest neighbor, case-based reasoning).

9.5 Write an algorithm for *k-nearest-neighbor classification* given *k*, the nearest number of neighbors, and *n*, the number of attributes describing each tuple.

9.6 Briefly describe the classification processes using (a) *genetic algorithms*, (b) *rough sets*, and (c) *fuzzy sets*.

9.7 Example 9.3 showed a use of error-correcting codes for a *multiclass classification* problem having four classes.

(a) Suppose that, given an unknown tuple to label, the seven trained binary classifiers collectively output the codeword 0101110, which does not match a codeword for any of the four classes. Using error correction, what class label should be assigned to the tuple?

(b) Explain why using a 4-bit vector for the codewords is insufficient for error correction.

9.8 *Semi-supervised classification*, *active learning*, and *transfer learning* are useful for situations in which unlabeled data are abundant.

(a) Describe *semi-supervised classification*, *active learning*, and *transfer learning*. Elaborate on applications for which they are useful, as well as the challenges of these approaches to classification.

(b) Research and describe an approach to semi-supervised classification other than self-training and cotraining.

(c) Research and describe an approach to active learning other than pool-based learning.

(d) Research and describe an alternative approach to instance-based transfer learning.

9.10 Bibliographic Notes

For an introduction to Bayesian belief networks, see Darwiche [Dar10] and Heckerman [Hec96]. For a thorough presentation of probabilistic networks, see Pearl [Pea88] and Koller and Friedman [KF09]. Solutions for learning the belief network structure from training data given observable variables are proposed in Cooper and Herskovits [CH92]; Buntine [Bun94]; and Heckerman, Geiger, and Chickering [HGC95]. Algorithms for inference on belief networks can be found in Russell and Norvig [RN95] and Jensen [Jen96]. The method of gradient descent, described in Section 9.1.2, for training Bayesian belief networks, is given in Russell, Binder, Koller, and Kanazawa [RBKK95]. The example given in Figure 9.1 is adapted from Russell et al. [RBKK95].

Alternative strategies for learning belief networks with hidden variables include application of Dempster, Laird, and Rubin's [DLR77] EM (Expectation Maximization) algorithm (Lauritzen [Lau95]) and methods based on the minimum description length

principle (Lam [Lam98]). Cooper [Coo90] showed that the general problem of inference in unconstrained belief networks is NP-hard. Limitations of belief networks, such as their large computational complexity (Laskey and Mahoney [LM97]), have prompted the exploration of hierarchical and composable Bayesian models (Pfeffer, Koller, Milch, and Takusagawa [PKMT99] and Xiang, Olesen, and Jensen [XOJ00]). These follow an object-oriented approach to knowledge representation. Fishelson and Geiger [FG02] present a Bayesian network for genetic linkage analysis.

The perceptron is a simple neural network, proposed in 1958 by Rosenblatt [Ros58], which became a landmark in early machine learning history. Its input units are randomly connected to a single layer of output linear threshold units. In 1969, Minsky and Papert [MP69] showed that perceptrons are incapable of learning concepts that are linearly inseparable. This limitation, as well as limitations on hardware at the time, dampened enthusiasm for research in computational neuronal modeling for nearly 20 years. Renewed interest was sparked following the presentation of the backpropagation algorithm in 1986 by Rumelhart, Hinton, and Williams [RHW86], as this algorithm can learn concepts that are linearly inseparable.

Since then, many variations of backpropagation have been proposed, involving, for example, alternative error functions (Hanson and Burr [HB87]); dynamic adjustment of the network topology (Mézard and Nadal [MN89]; Fahlman and Lebiere [FL90]; Le Cun, Denker, and Solla [LDS90]; and Harp, Samad, and Guha [HSG90]); and dynamic adjustment of the learning rate and momentum parameters (Jacobs [Jac88]). Other variations are discussed in Chauvin and Rumelhart [CR95]. Books on neural networks include Rumelhart and McClelland [RM86]; Hecht-Nielsen [HN90]; Hertz, Krogh, and Palmer [HKP91]; Chauvin and Rumelhart [CR95]; Bishop [Bis95]; Ripley [Rip96]; and Haykin [Hay99]. Many books on machine learning, such as Mitchell [Mit97] and Russell and Norvig [RN95], also contain good explanations of the backpropagation algorithm.

There are several techniques for extracting rules from neural networks, such as those found in these papers: [SN88, Gal93, TS93, Avn95, LSL95, CS96, LGT97]. The method of rule extraction described in Section 9.2.4 is based on Lu, Setiono, and Liu [LSL95]. Critiques of techniques for rule extraction from neural networks can be found in Craven and Shavlik [CS97]. Roy [Roy00] proposes that the theoretical foundations of neural networks are flawed with respect to assumptions made regarding how connectionist learning models the brain. An extensive survey of applications of neural networks in industry, business, and science is provided in Widrow, Rumelhart, and Lehr [WRL94].

Support Vector Machines (SVMs) grew out of early work by Vapnik and Chervonenkis on statistical learning theory [VC71]. The first paper on SVMs was presented by Boser, Guyon, and Vapnik [BGV92]. More detailed accounts can be found in books by Vapnik [Vap95, Vap98]. Good starting points include the tutorial on SVMs by Burges [Bur98], as well as textbook coverage by Haykin [Hay08], Kecman [Kec01], and Cristianini and Shawe-Taylor [CS-T00]. For methods for solving optimization problems, see Fletcher [Fle87] and Nocedal and Wright [NW99]. These references give additional details alluded to as "fancy math tricks" in our text, such as transformation of the problem to a Lagrangian formulation and subsequent solving using Karush-Kuhn-Tucker (KKT) conditions.

For the application of SVMs to regression, see Schölkopf, Bartlett, Smola, and Williamson [SBSW99] and Drucker, Burges, Kaufman, Smola, and Vapnik [DBK+97]. Approaches to SVM for large data include the sequential minimal optimization algorithm by Platt [Pla98], decomposition approaches such as in Osuna, Freund, and Girosi [OFG97], and CB-SVM, a microclustering-based SVM algorithm for large data sets, by Yu, Yang, and Han [YYH03]. A library of software for support vector machines is provided by Chang and Lin at *www.csie.ntu.edu.tw/∼cjlin/libsvm/*, which supports multiclass classification.

Many algorithms have been proposed that adapt frequent pattern mining to the task of classification. Early studies on associative classification include the CBA algorithm, proposed in Liu, Hsu, and Ma [LHM98]. A classifier that uses *emerging patterns* (itemsets with support that varies significantly from one data set to another) is proposed in Dong and Li [DL99] and Li, Dong, and Ramamohanarao [LDR00]. CMAR is presented in Li, Han, and Pei [LHP01]. CPAR is presented in Yin and Han [YH03b]. Cong, Tan, Tung, and Xu describe RCBT, a method for mining top-k covering rule groups for classifying high-dimensional gene expression data with high accuracy [CTTX05].

Wang and Karypis [WK05] present HARMONY (Highest confidence classificAtion Rule Mining fOr iNstance-centric classifYing), which directly mines the final classification rule set with the aid of pruning strategies. Lent, Swami, and Widom [LSW97] propose the ARCS system regarding mining multidimensional association rules. It combines ideas from association rule mining, clustering, and image processing, and applies them to classification. Meretakis and Wüthrich [MW99] propose constructing a naïve Bayesian classifier by mining long itemsets. Veloso, Meira, and Zaki [VMZ06] propose an association rule–based classification method based on a lazy (noneager) learning approach, in which the computation is performed on a demand-driven basis.

Studies on discriminative frequent pattern–based classification were conducted by Cheng, Yan, Han, and Hsu [CYHH07] and Cheng, Yan, Han, and Yu [CYHY08]. The former work establishes a theoretical upper bound on the discriminative power of frequent patterns (based on either information gain [Qui86] or Fisher score [DHS01]), which can be used as a strategy for setting minimum support. The latter work describes the DDPMine algorithm, which is a direct approach to mining discriminative frequent patterns for classification in that it avoids generating the complete frequent pattern set. H. Kim, S. Kim, Weninger, et al. proposed an NDPMine algorithm that performs frequent and discriminative pattern–based classification by taking *repetitive* features into consideration [KKW+10].

Nearest-neighbor classifiers were introduced in 1951 by Fix and Hodges [FH51]. A comprehensive collection of articles on nearest-neighbor classification can be found in Dasarathy [Das91]. Additional references can be found in many texts on classification, such as Duda, Hart, and Stork [DHS01] and James [Jam85], as well as articles by Cover and Hart [CH67] and Fukunaga and Hummels [FH87]. Their integration with attribute weighting and the pruning of noisy instances is described in Aha [Aha92]. The use of search trees to improve nearest-neighbor classification time is detailed in Friedman, Bentley, and Finkel [FBF77]. The partial distance method was proposed by researchers in vector quantization and compression. It is outlined in Gersho and Gray

[GG92]. The editing method for removing "useless" training tuples was first proposed by Hart [Har68].

The computational complexity of nearest-neighbor classifiers is described in Preparata and Shamos [PS85]. References on case-based reasoning include the texts by Riesbeck and Schank [RS89] and Kolodner [Kol93], as well as Leake [Lea96] and Aamodt and Plazas [AP94]. For a list of business applications, see Allen [All94]. Examples in medicine include CASEY by Koton [Kot88] and PROTOS by Bareiss, Porter, and Weir [BPW88], while Rissland and Ashley [RA87] is an example of CBR for law. CBR is available in several commercial software products. For texts on genetic algorithms, see Goldberg [Gol89], Michalewicz [Mic92], and Mitchell [Mit96].

Rough sets were introduced in Pawlak [Paw91]. Concise summaries of rough set theory in data mining include Ziarko [Zia91] and Cios, Pedrycz, and Swiniarski [CPS98]. Rough sets have been used for feature reduction and expert system design in many applications, including Ziarko [Zia91], Lenarcik and Piasta [LP97], and Swiniarski [Swi98]. Algorithms to reduce the computation intensity in finding reducts have been proposed in Skowron and Rauszer [SR92]. Fuzzy set theory was proposed by Zadeh [Zad65, Zad83]. Additional descriptions can be found in Yager and Zadeh [YZ94] and Kecman [Kec01].

Work on multiclass classification is described in Hastie and Tibshirani [HT98], Tax and Duin [TD02], and Allwein, Shapire, and Singer [ASS00]. Zhu [Zhu05] presents a comprehensive survey on semi-supervised classification. For additional references, see the book edited by Chapelle, Schölkopf, and Zien [CSZ06]. Dietterich and Bakiri [DB95] propose the use of error-correcting codes for multiclass classification. For a survey on active learning, see Settles [Set10]. Pan and Yang present a survey on transfer learning [PY10]. The TrAdaBoost boosting algorithm for transfer learning is given in Dai, Yang, Xue, and Yu [DYXY07].