Module 4

Rule based classification- 1R. Neural Networks-Back propagation. Support Vector Machines, Lazy Learners-K Nearest Neighbor Classifier. Accuracy and error Measures evaluation. Prediction:-Linear Regression and Non-Linear Regression

RULE-BASED ALGORITHMS

Rules are a good way of representing information or bits of knowledge. A rule-based classifier uses a set of IF-THEN rules for classification.

An IF-THEN rule is an expression of the form IF condition THEN conclusion.

An example is rule R1,

R1: IF age = youth AND student = yes THEN buys computer = yes. The "IF" part (or left side) of a rule is known as the rule antecedent or precondition. The "THEN" part (or right side) is the rule consequent. In the rule antecedent, the condition consists of one or more attribute tests (e.g., age = youth and student = yes) that are logically ANDed. The rule's consequent contains a class prediction (in this case, we are predicting whether a customer will buy a computer).

R1 can also be written as R1: (age = youth) $\wedge$ (student = yes) $\Rightarrow$ (buys computer = yes).

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

One straightforward way to perform classification is to generate if-then rules that cover all cases.

For example, we could have the following rules to determine classification of grades:

If 90 ≤ grade, then class class=A

If 80 ≤ grade and grade < 90, then class = B

If 70 ≤ S grade and grade < 80, then class = C

If 60 ≤ S grade and grade < 70, then class =D

If grade < 60, then class = F

A classification rule, r = (a, c), consists of the if or antecedent, a, part and the then or consequent portion, c. The antecedent contains a predicate that can be evaluated as true or false against each tuple in the database (and obviously in the training data). These rules relate directly to the corresponding DT that could be created. A DT can always be used to generate rules, but they are not equivalent. There are differences between rules and trees :

- The tree has an implied order in which the splitting is performed. Rules have no order.

- A tree is created based on looking at all classes. When generating rules, only one class must be examined at a time.

There are algorithms that generate rules from trees as well as algorithms that generate rules without first creating DTs.

1R Classification

These techniques are sometimes called covering algorithms because they attempt to generate rules exactly cover a specific class [WFOO]. Tree algorithms work in a topdown divide and conquer approach, but this need not be the case for covering algorithms. They generate the best rule possible by optimizing the desired classification probability. Usually the "best" attribute-value pair is chosen, as opposed to the best attribute with the tree-based algorithms. Suppose that we wished to generate a rule to classify persons as tall. The basic format for the rule is then

If ? then class = tall

The objective for the covering algorithms is to replace the "?" in this statement with predicates that can be used to obtain the "best" probability of being tall.

One simple approach is called 1R because it generates a simple set of rules that are equivalent to a DT with only one level. The basic idea is to choose the best attribute to perform the classification based on the training data. "Best" is defined here by counting the number of errors. In Table 4.4 this approach is illustrated using the height example, Outputl. If we only use the gender attribute, there are a total of 6/15 errors whereas if we use the height attribute, there are only 1/ 15. Thus, the height would be chosen and the six rules stated in the table would be used. As with ID3, 1R tends to choose attributes with a large number of values leading to overfitting.

lR can handle missing data by adding an additional attnbute value for the value of missing. Algorithm 4.10, which is adapted from [WFOO], shows the outline for this algorithm.

TABLE 4.4: 1R Classification

Option	Attribute	Rules	Errors	Total Errors
1	Gender	F → Medium	3/9	6/15
		M → Tall	3/6	
2	Height	(0, 1.6] → Short	0/2	1/15
		(1.6, 1.7] → Short	0/2	
		(1.7, 1.8] → Medium	0/3	
		(1.8, 1.9] → Medium	0/4	
		(1.9, 2.0] → Medium	1/2	
		(2.0, ∞) → Tall	0/2	

ALGORITHM 4.10

Input :

D / / Training data

R / /Attributes to consider for rules

C //Classes

Output :

R //Rules

lR algorithm:

//lR algorithm generates rules based on one attribute

R=Ø ;

for each A € R

do R_A = Ø ;

for each possible value , v, of A do

//v may be a range rather than a specific value

for each Cj € C find count (Cj);

// Here count is the number of occurrences of this class for this attribute

let Cm be the class with the largest count ;

$R_A = R_A$ U ((A = v) → (class = Cm)) ;

ERR_A = number of tuples incorrectly classifi ed by R_A

R = R_A where ERR_A is minimum ;

Classification by Backpropagation

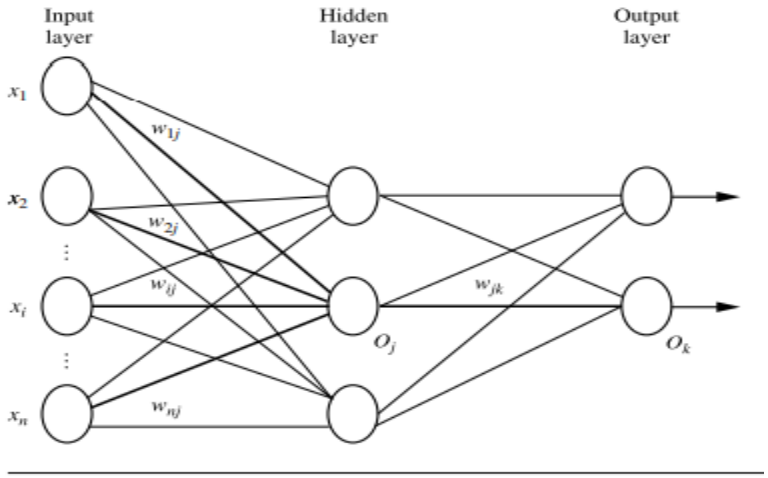
What is backpropagation?" Backpropagation is a neural network learning algorithm. The neural networks field was originally kindled by psychologists and neurobiologists who sought to develop and test computational analogs of neurons. Roughly speaking, a neural network is a set of connected input/output 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.

A Multilayer Feed-Forward Neural Network



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.

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 . 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.

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 below. The steps involved are expressed in terms of inputs, outputs, and errors, and may seem awkward if this is your first look at 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.

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 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) } }

- 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, 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 ,its output, Oj , is equal to its input value, Ij . 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

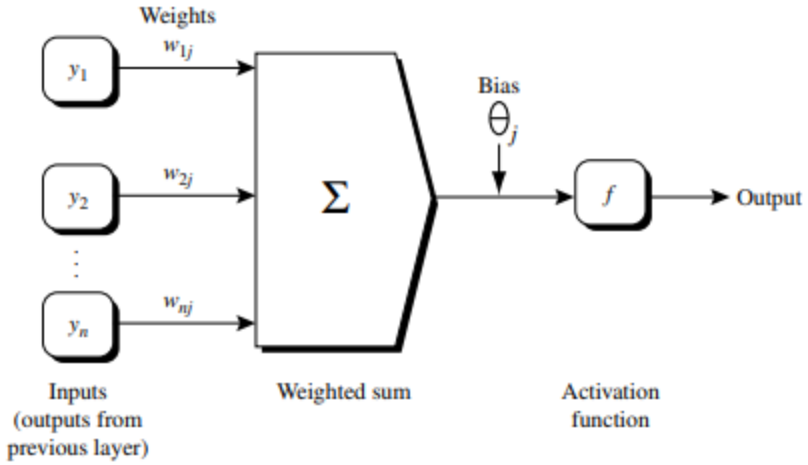


Figure 9.4 Hidden or output layer unit *j*: The inputs to unit *j* are outputs from the previous layer. These are multiplied by their corresponding weights to form a weighted sum, which is added to the bias associated with unit *j*. A nonlinear activation function is applied to the net input. (For ease 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 $(x_1, x_2, \ldots, x_n)$.)

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, Ij , to unit j is

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

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

Each unit in the hidden and output layers takes its net input and then applies an activation function to it, 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 Ij to unit j, then Oj , the output of unit j, is computed as

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

This function is also referred to as a squashing function, because it maps a large input domain onto the smaller range of 0 to 1. The logistic function is nonlinear and differentiable, allowing the backpropagation algorithm to model classification problems that are linearly inseparable. We compute the output values, Oj , for each hidden layer, up to and including the output layer, which gives the network's prediction. In practice, it is a good idea to cache (i.e., save) the intermediate output values at each unit as they are required again later when backpropagating the error. This trick can substantially reduce the amount of computation required.

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

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

where Oj is the actual output of unit j, and Tj is the known target value of the given training tuple. Note that Oj(1 − Oj) is the derivative of the logistic function.

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

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

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 Δwij is the change in weight wij

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

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

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 meansquared distance between the network's class prediction and the known target value of the tuples.

Biases are updated by the following equations, where Δθj is the change in bias θj :

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

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

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.

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, X = (1, 0, 1), with a class label of 1. This example shows the calculations for backpropagation, given the first training tuple, 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.

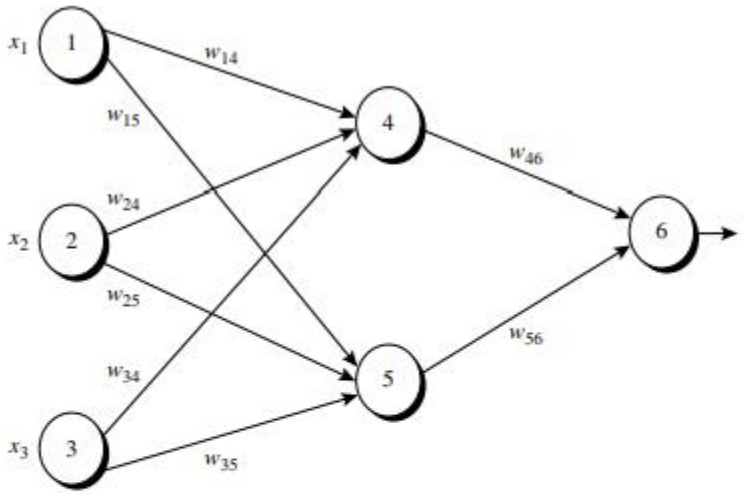


Figure 9.5 Example of a multilayer feed-forward neural network.

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, 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 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.

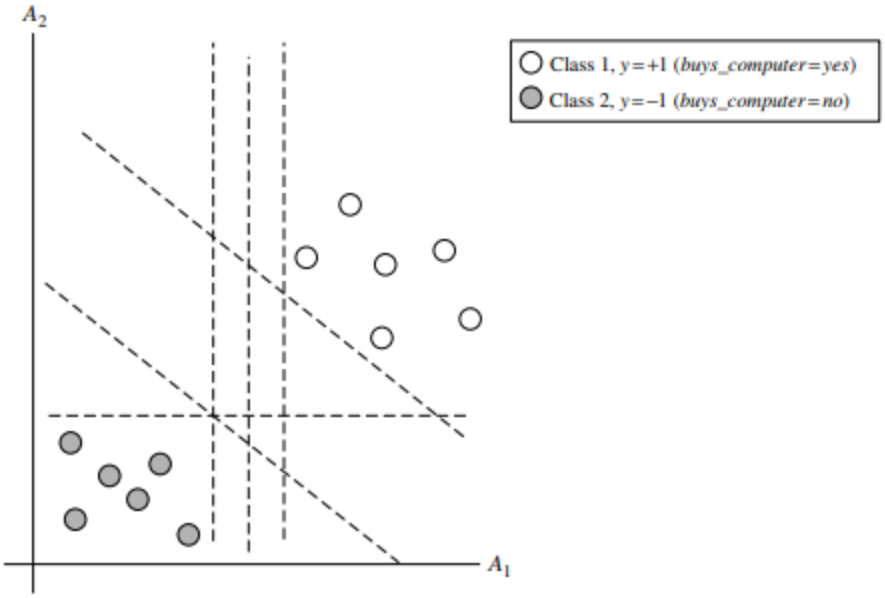
Support Vector Machines

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).

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.

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 (**X**1, y1), (**X**2, y2), ... , ($X_{|D|}$, $y_{|D|}$), where **Xi** is the set of training tuples with associated class labels, yi .

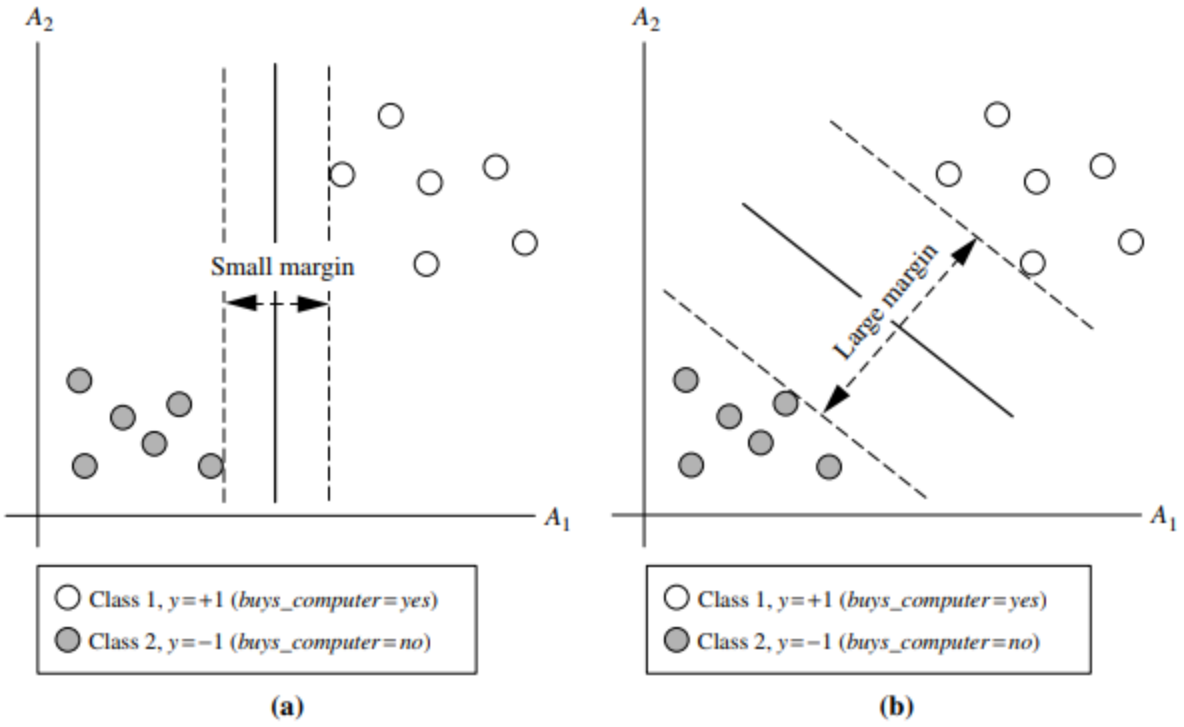


The 2-D training data are linearly separable. There are an infinite number of possible separating hyperplanes or "decision boundaries," some of which are shown here as dashed lines. Which one is best?

Each yi can take one of two values, either +1 or −1 (i.e., yi ∈ {+1, − 1}). 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, A1 and A2, 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.



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

$$\mathbf{W} \cdot \mathbf{X} + b = 0$$

where W is a weight vector, namely, W = {w1, w2,..., wn}; 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, A1 and A2, as in Figure 9.8(b). Training tuples are 2-D (e.g., X = (x1, x2)), where x1 and x2 are the values of attributes A1 and A2, respectively, for X. If we think of b as an additional weight, w0, we can rewrite Eq. (9.12) as

$$w_0 + w_1 x_1 + w_2 x_2 = 0. \quad (9.13)$$

Thus, any point that lies above the separating hyperplane satisfies

$$w_0 + w_1 x_1 + w_2 x_2 > 0. \quad (9.14)$$

Similarly, any point that lies below the separating hyperplane satisfies

$$w_0 + w_1 x_1 + w_2 x_2 < 0. \quad (9.15)$$

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

can be written as

$H1 : w_0 + w_1x_1 + w_2x_2 \geq 1$ for $y_i = +1$

$H2 : w_0 + w_1x_1 + w_2x_2 \leq -1$ for $y_i = -1$

That is, any tuple that falls on or above $H1$ belongs to class +1, and any tuple that falls

on or below $H2$ belongs to class -1. Combining the two inequalities of Eqs.

, we get

$y_i(w_0 + w_1x_1 + w_2x_2) \geq 1, \forall i.$ (9.18)

Any training tuples that fall on hyperplanes H1 or H2 (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.

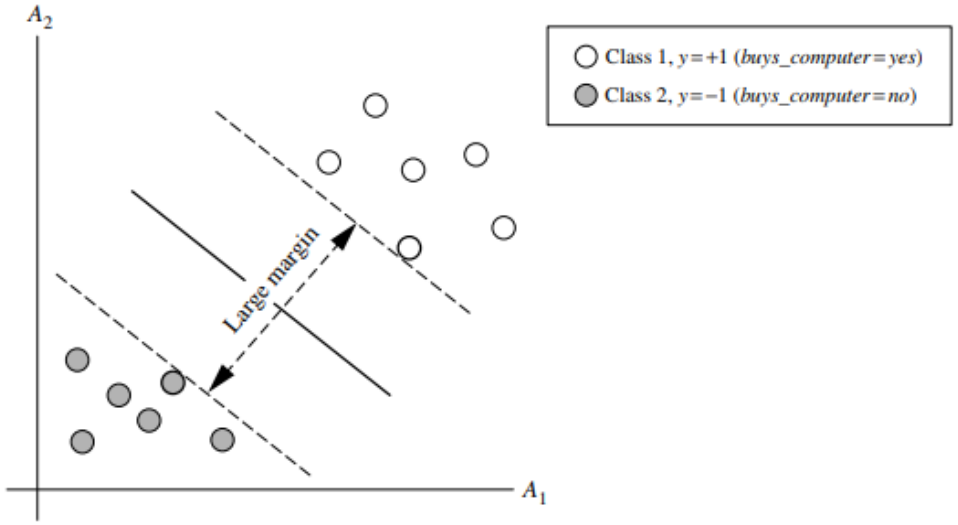


Figure 9.9 Support vectors. The SVM finds the maximum separating hyperplane, that is, the one with maximum distance between the nearest training tuples. The support vectors are shown with a thicker border.

The distance from the separating hyperplane to any point on H1 is $\frac{1}{||W||}$, where ||W|| is the Euclidean norm of W, that is, $\sqrt{(W.W)^2}$ By definition, this is equal to the distance from any point on H2 to the separating hyperplane. Therefore, the maximal margin is $\frac{2}{||W||}$.

"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

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

where yi is the class label of support vector Xi ; X T is a test tuple; αi and b0 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 αi 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, 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 X^T falls on or above the MMH, and so the SVM predicts that X^T belongs to class +1 (representing buys computer = yes, in our case). If the sign is negative, then X^T falls on or below the MMH and the class prediction is −1 (representing buys computer = no).

The Case When the Data Are Linearly Inseparable

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.

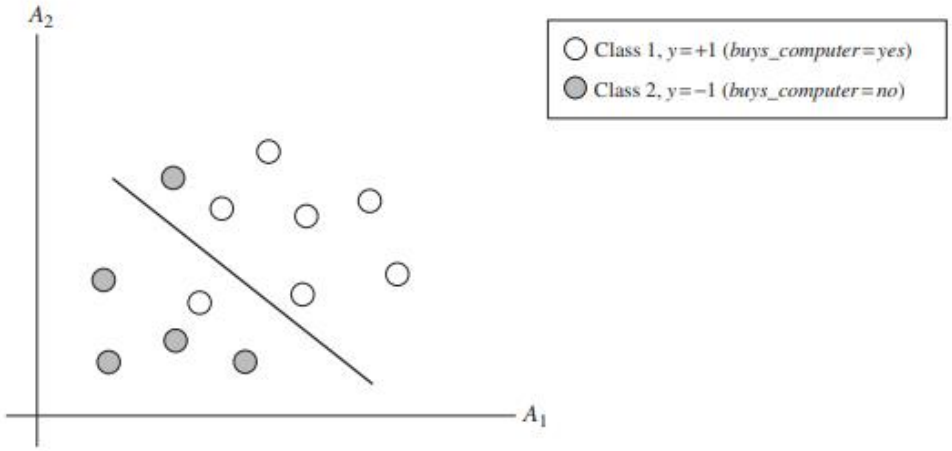


Figure 9.10 A simple 2-D case showing linearly inseparable data. Unlike the linear separable data of Figure 9.7, here it is not possible to draw a straight line to separate the classes. Instead, the decision boundary is nonlinear.

Example 9.2

Nonlinear transformation of original input data into a higher dimensional space. Consider the following example. A 3-D input vector X = (x1, x2, x3) is mapped into a 6-D space, Z, using the mappings ϕ1(X) = x1, ϕ2(X) = x2, ϕ3(X) = x3, ϕ4(X) = (x1) 2 , ϕ5(X) = x1x2, and ϕ6(X) = x1x3. A decision hyperplane in the new space is d(Z) = WZ + b, where W and Z are vectors. This is linear. We solve for W and b and then substitute back so that the linear decision hyperplane in the new (Z) space corresponds to a nonlinear second-order polynomial in the original 3-D input space:

$d(Z) = w1x1 + w2x2 + w3x3 + w4 (x1)^2 + w5x1x2 + w6x1x3 + b$

$= w1z1 + w2z2 + w3z3 + w4z4 + w5z5 + w6z6 + b$

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, 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, ϕ(Xi)· ϕ(Xj), where ϕ(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(Xi , Xj), to the original input data. That is,

K(Xi , Xj) = ϕ(Xi)· ϕ(Xj). (9.20)

In other words, everywhere that ϕ(Xi)· ϕ(Xj) appears in the training algorithm, we can replace it with K(Xi ,Xj). 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, αi . 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 have been studied.

Three admissible kernel functions are

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

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

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

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 .

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.

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 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 the example of lazy learner: k-nearest-neighbor classifiers

k-Nearest-Neighbor Classifiers

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

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

"Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, X1 = (x11, x12,..., x1n) and X2 = (x21, x22,..., x2n), is

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

In other words, for each numeric attribute, we take the difference between the corresponding values of that attribute in tuple X1 and in tuple X2, square this difference, and accumulate it. The square root is taken of the total accumulated distance count. Typically, we normalize the values of each attribute before using Eq. (9.22). This helps prevent attributes with initially large ranges (e.g., income) from outweighing attributes with initially smaller ranges (e.g., binary attributes). Min-max normalization, for example, can be used to transform a value v of a numeric attribute A to v 0 in the range [0, 1] by computing

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

where minA and maxA 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 X1 with that in tuple X2. If the two are identical (e.g., tuples X1 and X2 both have the color blue), then the difference between the two is taken as 0. If the two are different (e.g., tuple X1 is blue but tuple X2 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 X1 and/or in tuple X2, 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 X1 and X2, then the difference is also taken to be 1. If only one value is missing and the other (which we will call v 0) is present and normalized, then we can take the difference to be either $|1 - v'|$ or $|0 - v'|$ (i.e., $1 - v'$ or v'), whichever is greater.

“How can I determine a good value for k, the number of neighbors?” This can be determined experimentally. Starting with k = 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, 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.

Prediction

What if we would like to predict a continuous value, rather than a categorical label?" The prediction of continuous values can be modeled by statistical techniques of regression. For example, we may like to develop a model to predict the salary of college graduates with 10 years of work experience, or the potential sales of a new product given its price. Many problems can be solved by linear regression, and even more can betackled by applying transformations to the variables so that a nonlinear problem can be converted to a linear one.

Linear and multiple regression

What is linear regression?" In linear regression, data are modeled using a straight line. Linear regression is the simplest form of regression. Bivariate linear regression models a random variable, Y (called a response variable), as a linear function of another random variable, X (called a predictor variable), i.e.,

$$Y = \alpha + \beta X$$

where the variance of Y is assumed to be constant, and α and β are regression coeffcients specifying the Y intercept and slope of the line, respectively. These coeffcients can be solved for by the method of least squares, which minimizes the error between the actual line separating the data and the estimate of the line. Given s samples or data points of the form (x1,y1), (x2, y2), .., (xs, ys), then the regression coeffcients can be estimated using this method with Equations (7.23) and (7.24),

$$\beta = \frac{\sum_{i=1}^{s}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{s}(x_i - \bar{x})^2}, \tag{7.23}$$

$$\alpha = \bar{y} - \beta\bar{x}, \tag{7.24}$$

where $\bar{x}$ is the average of x1, x2,... xs, and $\bar{y}$ is the average of y1,y2,... ys. The coeffcients α and β often provide good approximations to otherwise complicated regression equations.

Example 7.6 Linear regression using the method of least squares. Table 7.7 shows a set of paired data where X is the number of years of work experience of a college graduate and Y is the corresponding salary of the graduate. A plot of the data is shown in Figure 7.16, suggesting a linear relationship between the two variables, X and Y. We model the relationship that salary may be related to the number of years of work experience with the equation $Y = \alpha + \beta X$

X *years experience*	Y *salary (in $1000)*
3	30
8	57
9	64
13	72
3	36
6	43
11	59
21	90
1	20
16	83

Table 7.7: Salary data.

Given the above data, we compute $\bar{x}$ =9.1and $\bar{y}$= 55.4. Substituting these values into Equation (7.23), we get

$$\beta = \frac{(3-9.1)(30-55:4)+(8-9:1)(57-55:4)+\ldots+(16-9.1)(83-55.4)}{(3-9.1)2+(8-9.1)2+\ldots+(16-9.1)2}$$

= 3.7

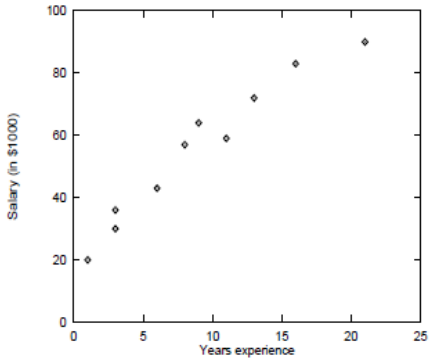


Figure 7.16: Plot of the data in Table 7.7 for Example 7.6. Although the points do not fall on a straight line, the overall pattern suggests a linear relationship between X (*years experience*) and Y (*salary*).

$$\alpha = 55.4 - (3.7)(9.1) = 21.7$$

Thus, the equation of the least squares line is estimated by Y = 21.7 + 3.7X. Using this equation, we can predict that the salary of a college graduate with, say, 10 years of experience is $58.7K.

Multiple regression is an extension of linear regression involving more than one predictor variable. It allows response variable Y to be modeled as a linear function of a multidimensional feature vector. An example of a multiple regression model based on two predictor attributes or variables, X1 and X2, is shown in Equation (7.25).

$Y = \alpha + \beta 1X1 + \beta 2X2$ (7.25)

The method of least squares can also be applied here to solve for α, β1, and β2.

Nonlinear regression

How can we model data that does not show a linear dependence? For example, what if a given response variable and predictor variables have a relationship that may be modeled by a polynomial function?" Polynomial regression can be modeled by adding polynomial terms to the basic linear model. By applying transformations to the variables, we can convert the nonlinear model into a linear one that can then be solved by the method of least squares.

Transformation of a polynomial regression model to a linear regression model. Consider a cubic polynomial relationship given by Equation (7.26).

$$Y = \alpha + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 \quad (7.26)$$

To convert this equation to linear form, we define new variables as shown in Equation (7.27).

$$X_1 = X \qquad X_2 = X^2 \qquad X_3 = X^3 \quad (7.27)$$

Equation (7.26) can then be converted to linear form by applying the above assignments, resulting in the equation $Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3$, which is solvable by the method of least squares.

□

Some models are intractably nonlinear (such as the sum of exponential terms, for example) and cannot be converted to a linear model. For such cases, it may be possible to obtain least-square estimates through extensive calculations on more complex formulae.

Other regression models

Linear regression is used to model continuous-valued functions. It is widely used, owing largely to its simplicity.”Can it also be used to predict categorical labels?" Generalized linear models represent the theoretical foundation on which linear regression can be applied to the modeling of categorical response variables. In generalized linear models, the variance of the response variable Y is a function of the mean value of Y , unlike in linear regression, where the variance of Y is constant. Common types of generalized linear models include logistic regression and Poisson regression. Logistic regression models the probability of some event occurring as a linear function of a set of predictor variables. Count data frequently exhibit a Poisson distribution and are commonly modeled using Poisson regression.

Log-linear models approximate discrete multidimensional probability distributions. They may be used to estimate the probability value associated with data cube cells. For example, suppose we are given data for the attributes city, item, year, and sales. In the log-linear method, all attributes must be categorical, hence continuousvalued attributes (like sales) must frst be discretized. The method can then be used to estimate the probability of each cell in the 4-D base cuboid for the given attributes, based on the 2-D cuboids for

city and item, city and year, city and sales, and the 3-D cuboid for item, year, and sales. In this way, an iterative technique can be used to build higher order data cubes from lower order ones. The technique scales up well to allow for many dimensions. Aside from prediction, the log-linear model is useful for data compression (since the smaller order cuboids together typically occupy less space than the base cuboid) and data smoothing (since cell estimates in the smaller order cuboids are less subject to sampling variations than cell estimates in the base cuboid).

Classifer accuracy

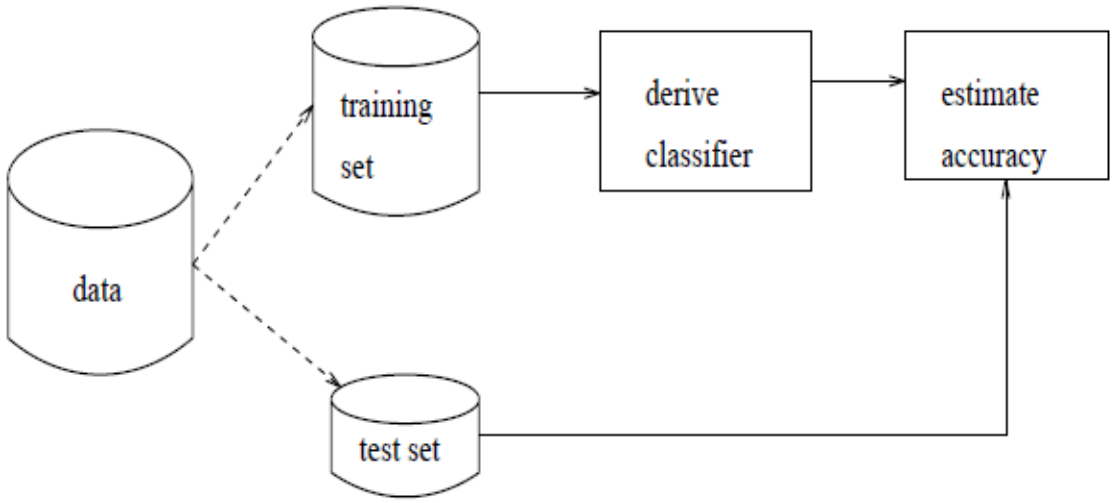


Figure 7.17: Estimating classifier accuracy with the holdout method.

Estimating classifer accuracy is important in that it allows one to evaluate how accurately a given classifer will correctly label future data, i.e., data on which the classifer has not been trained. For example, if data from previous sales are used to train a classifer to predict customer purchasing behavior, we would like some estimate of how accurately the classifer can predict the purchasing behavior of future customers. Accuracy estimates also help in the comparison of different classifers. In Section 7.9.1, we discuss techniques for estimating classifer accuracy, such as the holdout and k-fold cross-validation methods. Section 7.9.2 describes bagging and boosting, two strategies for increasing classifer accuracy. Section 7.9.3 discusses additional issues relating to classifer selection.

7.9.1 Estimating classifer accuracy

Using training data to derive a classifer and then to estimate the accuracy of the classifer can result in misleading over-optimistic estimates due to overspecialization of the learning algorithm (or model) to the data. Holdout and cross-validation are two common techniques for assessing classifer accuracy, based on randomly-sampled partitions of the given data. In the holdout method, the given data are randomly partitioned into two independent sets, a training set and a test set. Typically, two thirds of the data are allocated to the training set, and the remaining one third is allocated to the test set. The training set is used

to derive the classifer, whose accuracy is estimated with the test set (Figure 7.17). The estimate is pessimistic since only a portion of the initial data is used to derive the classifer.

Random subsampling is a variation of the holdout method in which the holdout method is repeated k times. The overall accuracy estimate is taken as the average of the accuracies obtained from each iteration. In k-fold cross validation, the initial data are randomly partitioned into k mutually exclusive subsets or \folds", S1, S2 .... Sk, each of approximately equal size. Training and testing is performed k times. In iteration i, the subset Si is reserved as the test set, and the remaining subsets are collectively used to train the classifer. That is, the classifer of the first iteration is trained on subsets S2 ... Sk, and tested on S1 , the classifer of the section iteration is trained on subsets S1; S3; ::; Sk, and tested on S2; and so on. The accuracy estimate is the overall number of correct classifcations from the k iterations, divided by the total number of samples in the initial data.

In stratifed cross-validation, the folds are stratifed so that the class distribution of the samples in each fold is approximately the same as that in the initial data.

Other methods of estimating classifer accuracy include bootstrapping, which samples the given training instances uniformly with replacement, and leave-one-out, which is k-fold cross validation with k set to s, the number of initial samples. In general, stratifed 10-fold cross-validation is recommended for estimating classifer accuracy (even if computation power allows using more folds) due to its relatively low bias and variance.

The use of such techniques to estimate classifer accuracy increases the overall computation time, yet is useful for selecting among several classifers.

7.9.2 Increasing classifer accuracy

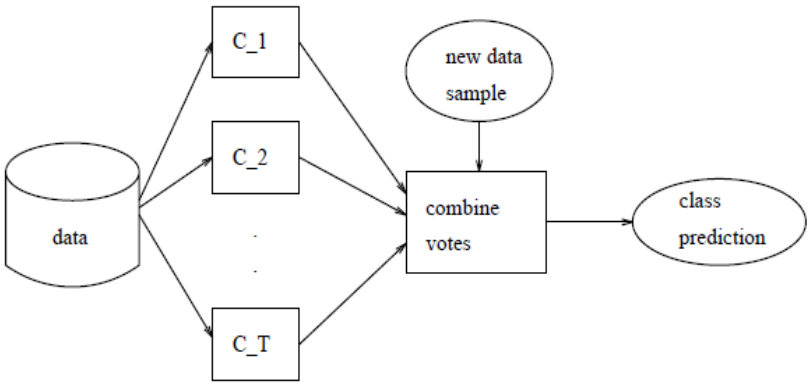


Figure 7.18: Increasing classifier accuracy: Bagging and boosting each generate a set of classifiers, $C_1, C_2, .., C_T$. Voting strategies are used to combine the class predictions for a given unknown sample.

In the previous section, we studied methods of estimating classifer accuracy. In Section 7.3.2, we saw how pruning can be applied to decision tree induction to help improve the accuracy of the resulting decision trees. Are there general techniques for improving classifer accuracy?

The answer is yes. Bagging (or boostrap aggregation) and boosting are two such techniques (Figure 7.18). Each combines a series of T learned classifers, C1; C2; ::;CT, with the aim of creating an improved composite classifer, C*.

“How do these methods work?" Suppose that you are a patient and would like to have a diagnosis made based on your symptoms. Instead of asking one doctor, you may choose to ask several. If a certain diagnosis occurs more than the others, you may choose this as the fnal or best diagnosis. Now replace each doctor by a classifer, and you have the intuition behind bagging. Suppose instead, that you assign weights to the value" or worth of each doctor's diagnosis, based on the accuracies of previous diagnoses they have made. The fnal diagnosis is then a combination of the weighted diagnoses. This is the essence behind boosting.

Let us have a closer look at these two techniques.

Given a set S of s samples, bagging works as follows. For iteration t (t = 1,2, ... T), a training set St is sampled with replacement from the original set of samples, S. Since sampling with replacement is used, some of the original samples of S may not be included in St, while others may occur more than once. A classifer Ct is learned for each training set, St. To classify an unknown sample, X, each classifer Ct returns its class prediction, which counts as one vote. The bagged classifer, Cf, counts the votes and assigns the class with the most votes to X. Bagging can be applied to the prediction of continuous values by taking the average value of each vote, rather than the majority.

In boosting, weights are assigned to each training sample. A series of classifers is learned. After a classifer Ct is learned, the weights are updated to allow the subsequent classifer, Ct+1, to “pay more attention" to the misclassifcation errors made by Ct. The fnal boosted classifer, C*, combines the votes of each individual classifer, where the weight of each classifer's vote is a function of its accuracy. The boosting algorithm can be extended for the prediction of continuous values.

7.9.3 Is accuracy enough to judge a classifer?

In addition to accuracy, classifers can be compared with respect to their speed, robustness (e.g., accuracy on noisy data), scalability, and interpretability. Scalability can be evaluated by assessing the number of I/O operations involved for a given classifcation algorithm on data sets of increasingly large size. Interpretability is subjective, although we may use objective measurements such as the complexity of the resulting classifer (e.g., number of tree nodes for decision trees, or number of hidden units for neural networks, etc.) in assessing it.

“Is it always possible to assess accuracy?" In classifcation problems, it is commonly assumed that all objects are uniquely classifable, i.e., that each training sample can belong to only one class. As we have discussed above, classifcation algorithms can then be compared according to their accuracy. However, owing to the wide diversity of data in large databases, it is not always reasonable to assume that all objects are uniquely classifable. Rather, it is more probable to assume that each object may belong to more than one class. How then, can the accuracy of classifers on large databases be measured? The accuracy measure is not appropriate, since it does not take into account the possibility of samples belonging to more than one class.

Rather than returning a class label, it is useful to return a probability class distribution. Accuracy measures may then use a second guess heuristic whereby a class prediction is judged as correct if it agrees with the frst or second most probable class. Although this does take into consideration, in some degree, the non-unique classifcation of objects, it is not a complete solution.

There have been numerous comparisons on the diferent classifcation methods, and the matter remains a research topic. No one method has been found to be superior over all others for all data sets. Issues such

as accuracy, training time, comprehensibility, and scalability must be considered and can involve trade-ofs, further complicating the quest for an overall superior method. Empirical studies show that the accuracy of many algorithms are sufciently similar that their diferences are statistically insignifcant, while training times may difer substantially (e.g., Lim, Loh, and Shih 2000). In general, most neural network and statistical classifcation methods involving splines tend to be more computationally intensive than most decision tree methods.