9
Pattern Recognition

Carl G. Looney
University of Nevada
Reno, Nevada

9.1 Introduction

9.1.1 Classification and Recognition

The conceptualization of things, or objects, as belonging to classes is at the core of all knowledge. We must caution, however, that classes are in the mind of the beholder and that there are often many possible attributes and ways to classify a set of objects. Commonly agreed upon classifications for many objects used routinely are a part of human cultures. We examine here some modern nonlinear methods for classifying and recognizing objects that include clustering, improved clustering, a newer type of fuzzy clustering, probabilistic and fuzzy neural networks, radial basis function neural networks, radial basis functional link nets, ellipsoidal basis function neural networks, ellipsoidal basis functional link nets, and fuzzy ellipsoidal classifiers.

Pattern recognition involves two processes: (1) **classification,** where a sample from a population of objects is partitioned into groups called **classes;** and (2) **recognition,** where a given unknown object from the same population is recognized as belonging to one of the established classes. Recognition is sometimes broken into recognition and **identification,** which means that a particular individual object is recognized. The terms *classification* and *recognition* are sometimes used interchangeably in the literature.

0-8493-1162-4/03/$0.00+$1.50

A classification process examines a sample of objects that represents a population of such objects and partitions it into subsets (the classes) according to similarity of the objects within classes and dissimilarity of the objects between classes. This type of process is called **self-organization, unsupervised learning,** or **clustering** of the sample into **clusters** (classes or subclasses). On the other hand, once a process has clustered the sample into classes and each object is assigned a class **label**—an index value (or codeword) that designates the particular class—a **recognizer** can be trained to assign a class label to any unknown object from the same population (pattern recognition). The training process is called **supervised learning** or **training** of the recognizer. A trained recognizer can perform pattern recognition online.

9.1.2 Features, Vectors, and Prototypes

The objects to be recognized as belonging to one class or another have certain properties that we use to distinguish between classes. These properties (attributes) are the observables, where the observation provides a value for each of a set of properties. A fixed set of properties is used for a particular population and the set of their values for an object determines whether it belongs to a class or not. The individual properties are called **features** of the population. Suppose that there are N features for a population to be used for recognition. These N features are ordered into an N-tuple so that a set of observed values for an object forms a vector, called a **feature vector**. Thus, the feature vectors represent the objects in a population. Pattern recognition is done on feature vectors.

For example, suppose a type of beetle in a certain geographical region has three subtypes and that one is a voracious eater that will destroy the crops. One subtype has a gray-green back when mature, while the other two have light green and dark green backs when mature. Further, suppose that the gray-green beetles have shorter length and are wider than the light and dark green beetles and that the dark green ones have longer antennae. Let us choose these properties as the features and assign quantities to them. For example, for the first feature we could assign 1 to gray-green, 2 to light green, and 3 to dark green. We can also assign the width and length as the second and third features so that the feature vector for each observed beetle is then the 3-tuple (n, w, l).

In the real world the situation is usually more complex. Suppose that the younger beetles are not well differentiated as to color, so it is not a clear case of gray-green, light green, or dark green. Also, the younger beetles are smaller so that the width and height are both smaller numbers. One way to attain greater accuracy is to let $x = 0$ represent gray-green, $x = 0.5$ represent light green, and $x = 1.0$ represent dark green. The observer could then assign a value such as 0.3 to color that is slightly more light green than gray-green. To account for age in the width and length measurements the ratio $r = w/l$ of width to length represents a shape parameter that is more independent of age. If we now observe young beetles in the spring to obtain feature vectors $\{(x, r)\}$, we can use these measurements to recognize the subtype to determine whether or not eradication methods are necessary to save the crop, provided that we know the typical feature vector for each subtype.

Figure 9.1 shows a set of vectors in the plane that represents a sample of beetles. The typical vector of each subtype sample is a **prototype** vector that represents that subtype, or class, of beetles. Given a feature vector (x, r), we test it against each prototype to determine the one it most resembles. A metric is used such as Euclidean distance or maximum component magnitude distance so that the closest prototype to (x, r) determines the subtype of (x, r).

Figure 9.2 shows the steps in the development of a PR system. Let us sample a given population of objects and determine a set of features to distinguish its classes. Then we draw a representative random sample for classification and obtain the observations for each feature of each sampled object to obtain a sample of feature vectors. This sample represents the population of interest. Now we **classify** the feature vectors and assign a class label to each one. The labels can be satisfactorily assigned by humans at times but is usually done by computational algorithms. Self-organizing learning is often better in cases where the feature values are noisy or the classes are not well known. In many cases the number K of classes is not known, but in others it is known, such as $K = 2$ for a *defective* or *nondefective*

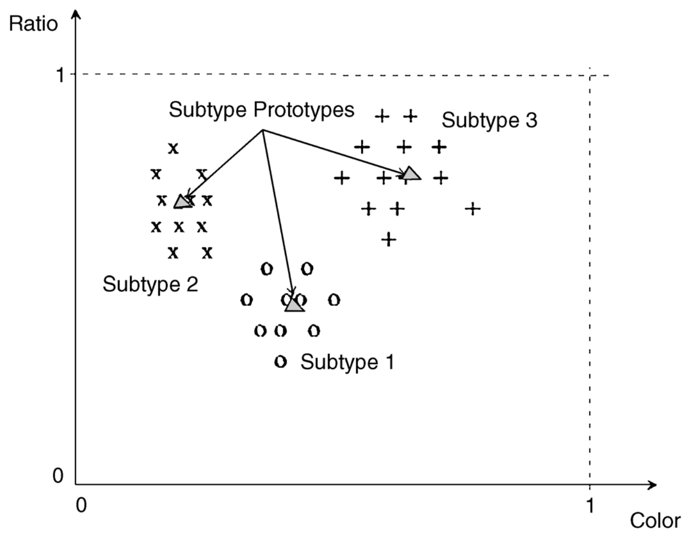

FIGURE 9.1 Feature vectors of observations on beetles.

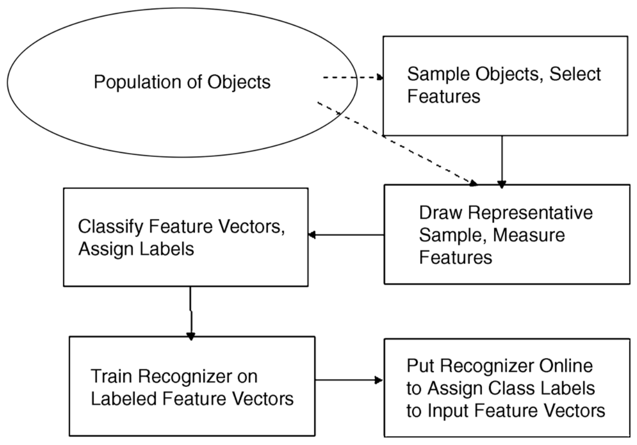
Population of Objects
Sample Objects, Select Features
Draw Representative Sample, Measure Features
Classify Feature Vectors, Assign Labels
Train Recognizer on Labeled Feature Vectors
Put Recognizer Online to Assign Class Labels to Input Feature Vectors

FIGURE 9.2 Development of a PR system.

product or for *malignant* or *benign* tumors, or $K = 3$ for *strong edge*, *weak edge* or *no edge* in image edge detection.

After classification and labeling of the sample feature vectors, the next step is to train a process via supervised learning to read any input feature vector from the same population and to output an accurate class label for it, that is, to *recognize* it. Recognition may require preprocessing raw data to extract special features, for example, taking the fast Fourier transform of a satellite image to obtain the power in certain frequency bands as texture feature values for either crop recognition or ocean wind speed from wave frequency patterns.

Feature selection is very important for classification and recognition [Looney, 1997]. If two features are strongly correlated, then they differentiate the same feature vectors and are redundant. Redundancy can be detected by performing a correlation of the features over the sample. We could eliminate one of each pair of correlated feature vectors to obtain a minimal set. An important principle is: *For every pair of classes, there must be at least one feature that separates them* (although it is preferable to have more than one such separating feature for each pair of classes).

9.2 Classification

9.2.1 Clustering with the *k*-Means Algorithm

Clustering is a self-organizing process that partitions an exemplar set of feature vectors into clusters (subsets) that represent classes. The ***k*-means** clustering algorithms are the simplest. These were developed by Forgy [1965] and MacQueen [1967]; we present the one by Forgy. Given a set of Q unlabeled feature vectors $\{\boldsymbol{x}^{(q)}: q = 1,\ldots,Q\}$, where each has N features, $x^{(q)} = (x_1^{(q)},\ldots,x_N^{(q)})$, we want to classify them into K clusters, where K is input by the user. The basic k-means algorithm is described below, where the **center,** or **prototype,** that represents a cluster is the average vector of that cluster.

Forgy's *k*-means Algorithm

Step 1: Randomly order the Q-feature vectors and input K (the number of classes).
Step 2: Select the first K of the Q-feature vectors as **seeds** (initial prototypes of classes).
Step 3: Assign each of the Q-feature vectors to the nearest prototype to form K classes (use the index $c[q] = k$ to designate that $\boldsymbol{x}^{(q)}$ belongs to Class k and count cluster sizes with $s[k]$, which is incremented every time a vector is assigned to Class k).
Step 4: Average the feature vectors in each class to find K new centers. To average all vectors in Class k, use the following technique, starting with $a[n][k] = 0.0$, where $a[n][k]$ is the value of component n of the Class k average vector and $x[n][q]$ is the nth component of $\boldsymbol{x}^{(q)}$.

```
for k = 1 to K do                          //For each Class k:
  for n = 1 to N do a[n][k] = 0.0;         //initialize averages
  for q = 1 to Q do                        //For given Class k, find
      if (c[q] = = k) then                 //all vectors in Cluster k
        for n = 1 to N do                  //and for each component n,
            a[n][k] = a[n][k] + x[n][q];   //sum that component
  if (s[k] > 1) then
      a[n][k] = a[n][k]/s[k];              //Average Cluster k
```

Step 5: *if ((not first pass) and (no class has changed)) then exit, else go to Step 3 above*

Many algorithms [Pena et al., 1999] use the k-means to start but apply some adjustment to aid with the seeding. They showed that random drawing of many more seeds worked better than the use of the first K-feature vectors. Selim and Ismail [1984] showed that the algorithm converges to a local minimum in the sum-squared error

$$E = \sum_{(k=1,K)} \sum_{\{c[q]=k\}} \left\{ \sum_{(n=1,N)} (x[n][q] - a[n][k])^2 \right\} \tag{9.1}$$

However, examples [Looney, 2002] showed that there is no guarantee of optimality of the clustering, which depends on K and the K initial seeds.

MacQueen's adjusted algorithm recomputes a new center every time a feature is assigned to a cluster. Snarey et al. [1997] proposed a maximum method for selecting a subset of the feature vectors for seeds. Kaufman and Rousseeuw [1990] used medians of clusters for the prototypes and then optimized a sum of squared distances (errors) in each cluster.

9.2.2 The Number *K* of Classes and Clustering Validity

The number of classes, K, may not be known, so we must reorder the Q-feature vectors and repeat the clustering several times to find the best clustering for estimated K, $K + 1$, $K - 1$, etc., to find a better clustering. But how do we know if a clustering is better? We can use a **clustering validity measure** as

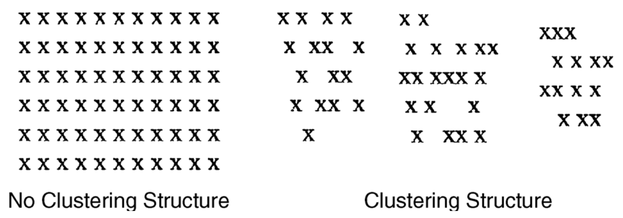

FIGURE 9.3 Structure in feature vector data.

described below. Validity measures in general are known to be problematic [Dubes and Jain, 1998], but it is more problematic when K is unknown to not use one and so we use the best one (see below).

A clustering is good if the clusters are relatively **compact** (packed closely about the center) and relatively **well separated** (no two centers are too close). Let σ_k^2 be the mean-square error (variance) of the kth cluster (fix k in Eq. 9.1 above). If these are relatively small for all k, then the clusters are compact, which is desirable. Let $D_{\min}$ be the minimum distance between all pairs of cluster *centers*, where a center is a single prototype for a cluster. It is desirable for this distance to be larger, rather than smaller, in which case the clusters are *well separated*. The **Xie-Beni** (XB) **clustering validity measure** is:

$$XB = (\sigma_1^2 + \cdots + \sigma_k^2)/D_{\min} \tag{9.2}$$

The smaller this measure is, the better is the clustering [Xie and Beni, 1991]. Thus, we move K in the direction that decreases XB until we get a minimum value for XB and accept the corresponding K. Figure 9.3 shows two sets of vectors, of which one has a moderately strong clustering structure.

There remains a problem with this algorithm. If two seeds are close together, then the result will be two clusters close together that should be merged into a single one, while a seed far from any other seeds can yield a large cluster that should be broken into two or more clusters. Ways to improve this have been found. For example, Chen and Wang [1999] used the *equalized universe* method, whereby a fine grid of initial seeds was used and a bell-shaped fuzzy-set membership function was centered on each with 25% overlap. But the number of such functions grows exponentially with the number of dimensions. The method of Chiu [1994] is a more efficient variant of the *mountain method* of Yager and Filev [1994], which sums Gaussians centered on the feature vectors to build a combined mountain function, which in turn is a variant of *Parzen windows* (summed Gaussians). However, these methods are all rather computationally expensive.

9.2.3 An Improved *k*-Means Algorithm

A simpler way to prevent bad clustering due to inadequate seeding is to modify the basic k-means algorithm to obtain the *improved k-means algorithm* [Looney, 2002]. We start with a large number of uniformly distributed seeds in the bounded N-dimensional feature space, but we reduce them considerably by eliminating those that are too close to another one. The test threshold is the average distance between seed vectors, which may be computed from a sample of seeds. Thus, we obtain a smaller number K of initial seeds that are uniformly distributed. Next, we assign each of the Q-feature vectors to a seed by minimum distance and then eliminate the empty clusters and any clusters of a size less than p (p is given by the user). Iteration now converges quickly to a lot of relatively small clusters. The closest ones are merged until the XB measure stops decreasing or until there are only two clusters remaining. The improved algorithm is given below.

The Improved *k*-Means Algorithm

Step 1: Draw a large number K of uniform random seed vectors $\mathbf{z}^{(k)}$ as initial centers.
Step 2: Eliminate all seed vectors that are too close to other seed vector and reduce K as needed.

```
for k1 = 1 to K − 1 do                      //For all except the last seed
  for k2 = k1 + 1 to K do                   //pair it with a different seed
    if (distance(k1,k2) < $\epsilon$) then          //If these two seeds are too close
      for k = k2 to K − 1 do                //then eliminate one by closing
        $\mathbf{z}^{(k)} = \mathbf{z}^{(k+1)}$;                 //up the indices over it
      K = K − 1;                            //and reducing no. clusters
```

Step 3: Assign each of the Q-feature vectors $\mathbf{x}^{(q)}$ to the nearest random seed vector by the assignment $c[q] = k$ and increment the size $s[k]$ of Class k

```
for k = 1 to K do s[k] = 0;                 //Initialize cluster sizes to 0
Dmin = 99999.9;                             //Initialize large min. distance
for q = 1 to Q do                           //For every feature vector q
  for k = 1 to K do                         //and every center seed k
    d = distance(k,q);                      //compute distance between them
    if (d < Dmin) then                      //If distance is smallest
      Dmin = d;                             //then save it and also save
      kmin = k;                             //its index
  c[q] = kmin;                              //Assign vector to nearest center
  s[kmin] = s[kmin] + 1;                    //and increase size of that cluster
```

Step 4: Eliminate all clusters that have fewer than p vectors and reduce K as needed, where $s[k]$ is the size of Cluster k and $c[q] = k$ means feature vector q belongs to Class k:

```
for k1 = 1 to K do                          //For each cluster:
  if (s[k1] < p) then                       //if cluster size is too small
    for k2 = k1 + 1 to K do                 //then eliminate that cluster
      for q = 1 to Q do                     //by finding its vectors and then
        if (c[q] = k2) then                 //(if vector q is in Class k2)
          c[q] = k2 − 1;                    //re-indexing them accordingly
      s[k2 − 1] = s[k2];                    //Also re-index sizes of clusters
      K = K − 1;                            //and reduce no. clusters
```

Step 5: Assign features to cluster centers, average cluster centers, and stop if no centers changed or else repeat this step.

Step 6: Compute the XB measure:

```
if ((this is not the first pass) and (XB increases)) then
  stop                                      //accept clustering of the previous iteration
else
  merge()                                   //merge two clusters with closest centers
  go to Step 5.
```

9.2.4 An Improved Fuzzy *k*-Means Algorithm

The next problem in clustering to be considered is the fact that outliers in a cluster can unduly affect the center, or prototype, when it is obtained as the cluster average (by averaging each component). Medians can be used in place of averages, although they may throw away good points as well as outliers. We use a type of fuzzy averaging here that puts the center prototype among the more densely situated points by using a weighted average. In the following algorithm we start with the improved *k*-means average of each cluster as the prototype, but then we center a Gaussian on that average prototype and compute fuzzy weights for a weighted average. We iterate this process until the fuzzy average does not change any further. We could also use medians or α-trimmed means [Bednar and Watt, 1984], where we throw away the α greatest and α least for each component and average the remainder component-wise.

A Weighted Fuzzy *k*-Means Averaging Algorithm

```
Step 1: Draw a large number $K$ of uniformly distributed random seed vectors in the feature space.
Step 2: Eliminate any seed vectors that are too close to other seed vectors and reduce $K$ accordingly
        (see Step 2 of the improved k-means algorithm above).
Step 3: Assign each of the Q-feature vectors $\boldsymbol{x}^{(q)}$ to the nearest random seed vector.
Step 4: Eliminate all seed vectors that are centers of empty clusters or have fewer than $p$ vectors, and
        reduce $K$ accordingly (see Step 4 of the improved k-means algorithm above).
Step 5: Compute the weighted fuzzy average of each class as the new class prototype with current
        $K$ (see the description of weighted fuzzy average below).
Step 6: Assign each of the Q-feature vectors to the class with the nearest weighted fuzzy average.
Step 7: If (first pass) or (any weighted fuzzy average has changed) then go to Step 5.
Step 8: Compute the XB measure (use Equation 9.1):
        if ((not the first pass) and (XB increases)) then
        stop                                    //accept clustering of previous iteration
Step 9: Merge the two clusters whose prototypes are closest and use the average of their two prototypes
        as a new prototype (seed) and reduce $K$ accordingly.
Step 10: Go to Step 5 (to find new class prototypes and reassignment, etc.)
```

The **weighted fuzzy average** (WFA) of the vectors in a cluster is done component-wise, so we explain it here for a single dimension. Let $\{x_1, \ldots, x_P\}$ be a set of P real numbers. To find its weighted fuzzy average we initially take the sample mean $\mu^{(0)}$ and variance σ^2 to start the process. We center a Gaussian over the current approximate WFA $\mu^{(r)}$ and iterate as follows:

$$w_p^{(r)} = \exp[-(x_p - \mu^{(r)})/2\sigma^2] / \textstyle\sum_{(m=1,P)} \exp[-(x_m - \mu^{(r)})/2\sigma^2] \tag{9.3}$$

$$\mu^{(r+1)} = \textstyle\sum_{(p=1,P)} w_p^{(r)} x_p, \quad r = 0, 1, 2, \ldots \tag{9.4}$$

The denominator in Eq. (9.3) standardizes the weights so they all sum to unity. We compute σ^2 on each of three or four iterations and then leave it fixed. After about five iterations the approximate WFA is sufficiently close to the true WFA. Schneider and Craig [1992] used a *weighted fuzzy expected value* for histogram adjustment, but it was based on a decaying exponential. We use Gaussians that are canonical fuzzy-set membership functions. Figure 9.4 below shows an example of five points (circles) that compares the mean, median, and the WFA.

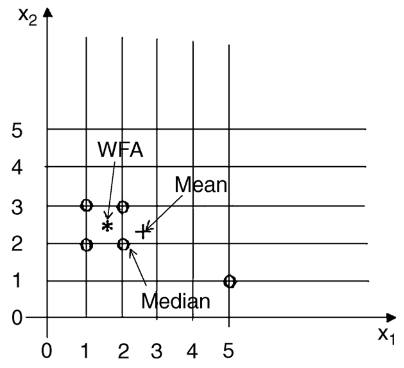

FIGURE 9.4 A mean, median, and WFA of five points.

We mention here that Bezdek's *fuzzy c-means* algorithm [Bezdek, 1973] computes weights for a weighted averaging of the vectors in a cluster. The solution for the weights optimizes a Lagrangian expression of the summed-squared error of the weighted average with respect to the centers, suitably constrained with Lagrangian multipliers. The solution weights are computed as

$$w_{qk} = \{1/\|\boldsymbol{x}^{(q)} - \boldsymbol{z}^{(k)}\|^2\}^{1/(p-1)} / \sum_{r=1,K} \{1/\|\boldsymbol{x}^{(q)} - \boldsymbol{z}^{(r)}\|^2\}^{1/(p-1)} \tag{9.5}$$

for $p > 1$ (p is usually 2 to 3 and remains fixed over all prototypes $\boldsymbol{z}^{(k)}$. The advantage was considered to be that every feature vector had a fuzzy weight for membership in each cluster (a new concept).

The performance of the fuzzy c-means clustering, however, can be very inadequate. It appears that smaller clusters should have a different value of p than larger clusters, i.e., a different weighting function, which is the case for our weighted fuzzy average with a different & value to fit each cluster that also evaluates the fuzzy-set membership of any feature vector in each class. The improved k-means performs well and is included in our weighted fuzzy clustering. We could also use medians or α-trimmed means [Bednar and Watt, 1984] as centers.

9.3 Recognition

Here we assume that we have a set of **exemplar** feature vectors (a sample of feature vectors that represent all classes of the population and that have been labeled by a classification process that assigns each feature vector a label to represent the class to which the vector belongs). The following methods train a recognition process via supervised learning for online recognition of an incoming stream of feature vectors from the population of interest. Of course, the process can learn only the labels that it is trained on in the supervised learning mode, so if the feature vectors are mislabeled or contain too much noise, then the trained process is inaccurate. However, a recognizer that is well trained on accurate data from representative exemplars can perform very well when put online.

9.3.1 Probabilistic Neural Networks

A probabilistic neural network (PNN) has three layers of nodes [Specht, 1988; 1990] (see [Anagnostopoulis et al., 2001] for an application to human face recognition). Figure 9.5 displays the architecture of a PNN that recognizes $K = 2$ classes, but it can be extended to any number K of classes. The input layer (on the left) contains N nodes: one for each of the N input features of a feature vector. These are fan-out nodes that branch at each feature input node to all nodes in the hidden (or middle) layer so that each hidden node receives the complete input feature vector $\boldsymbol{x}$. The hidden nodes are collected into groups, one group for each of the K classes, as shown in Figure 9.5. Each hidden node in the group for Class k corresponds

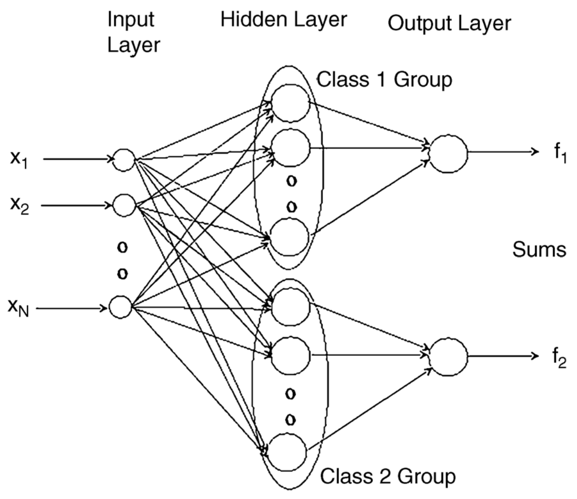

FIGURE 9.5 A two-class PNN.

to a Gaussian function centered on its associated feature vector in the *k*th class (there is a Gaussian for each exemplar feature vector). All of the Gaussians in a class group feed their functional values to the same output layer node for that class, so there are *K* output nodes (one for each class).

At the output node for Class *k*, all of the Gaussian values for Class *k* are summed and the total sum is scaled to force the integral of the sum to be unity so that the sum forms a probability density function. Here we temporarily use special notation for clarity. Let there be *P* exemplar feature vectors $\{\boldsymbol{x}^{(p)}: p = 1,\ldots,P\}$ labeled as Class 1, and let there be *Q* exemplar feature vectors $\{\boldsymbol{y}^{(q)}: q = 1,\ldots,Q\}$ labeled as Class 2 (see Figure 9.5). In the hidden layer there are *P* nodes in the group for Class 1 and *Q* nodes in the group for Class 2. The equations for each Gaussian centered on the respective Class 1 and Class 2 points $\boldsymbol{x}^{(p)}$ and $\boldsymbol{y}^{(q)}$ (feature vectors) are:

$$g_1(\boldsymbol{x}) = [1/\surd(2\pi\sigma^2)]\exp\{-\|\boldsymbol{x} - \boldsymbol{x}^{(p)}\|^2/(2\sigma^2)\} \tag{9.6}$$

$$g_2(\boldsymbol{y}) = [1/\surd(2\pi\sigma^2)]\exp\{-\|\boldsymbol{y} - \boldsymbol{y}^{(q)}\|^2/(2\sigma^2)\} \tag{9.7}$$

The σ values can be taken to be one half the average distance between the feature vectors in the same group, or at each exemplar it can be one half the distance from the exemplar to its nearest other exemplar vector. The *k*th output node sums the values received from the hidden nodes in the *k*th group, called *mixed Gaussians* or *Parzen windows*. The sums are defined by:

$$f_1(\boldsymbol{x}) = [1/\surd(2\pi\sigma^2)^P](1/P)\sum_{(p=1,P)} \exp\{-\|\boldsymbol{x} - \boldsymbol{x}^{(p)}\|^2/(2\sigma^2)\} \tag{9.8}$$

$$f_2(\boldsymbol{y}) = [1/\surd(2\pi\sigma^2)^Q](1/Q)\sum_{(q=1,Q)} \exp\{-\|\boldsymbol{y} - \boldsymbol{y}^{(q)}\|^2/(2\sigma^2)\} \tag{9.9}$$

Any input vector is put through both functions, and the maximum value (**maximum *a posteriori*,** or MAP value) of f_1 and f_2 decides the class. For $K > 2$ classes the process is analogous. There is no iteration or computation of weights. For a large number of Gaussians in a sum, the error can be significant (see [Wedding and Cios, 1998]). Thus, the feature vectors in each class may be reduced by thinning those that are too close to another one.

9.3.2 Fuzzy Neural Networks

Our **fuzzy neural networks** (FNNs) are similar to the PNNs. Let there be K classes, and let $\boldsymbol{x}$ be any feature vector from the population of interest to be recognized. The Class k exemplar feature vectors are denoted by $\boldsymbol{x}^{(qk)}$ for $qk = 1, \ldots, Qk$. We replace the functions of Eqs. (9.8) and (9.9) with the following more simply scaled sums.

$$\begin{aligned} f_1(\boldsymbol{x}) &= (1/Q_1)\textstyle\sum_{(p=1,P)} \exp\{-\|\boldsymbol{x} - \boldsymbol{x}^{(q1)}\|^2/(2\sigma^2)\} \\ &\vdots \qquad\qquad\qquad\qquad \vdots \qquad\qquad (9.10) \\ f_K(\boldsymbol{x}) &= (1/Q_K)\textstyle\sum_{(q=1,Q)} \exp\{-\|\boldsymbol{x} - \boldsymbol{x}^{(qK)}\|^2/(2\sigma^2)\} \end{aligned}$$

These functions are the K fuzzy-set membership functions whose functional values are the relative fuzzy truths of memberships in the K respective classes. Thus, $\boldsymbol{x}$ belongs to the class with the highest fuzzy value. When there is a clear winner, then $\boldsymbol{x}$ belongs to a single class, but otherwise it may belong to more than one class with the given relative fuzzy truths. No training of weights is required. The exemplar vectors may be thinned as previously done.

9.3.3 Radial Basis Function Neural Networks

These neural networks (NNs) have three layers: (1) the input layer of N nodes, where the respective N features in a feature vector are input; (2) the hidden (middle) layer, where at each node the input feature vector is put through a Gaussian **radial basis function** (RBF) that is centered on a corresponding exemplar vector $\boldsymbol{v}$, as shown in Figure 9.6; and (3) the output layer, where the inputs from all of the hidden nodes are combined at each output node in a weighted average. The weights $\{u_{mj}\}$ used in the weighted average are adjusted during the training so as to map each exemplar feature vector into its correct output codeword to approximately equal the **target** vector (desired output codeword that is a label). Figure 9.7 presents the architecture of a **radial basis function neural network** (RBFNN).

Again we let $\{\boldsymbol{x}^{(q)}: q = 1, \ldots, Q\}$ be a set of Q exemplar feature vectors for training, where each has N components.

$$x^{(q)} = (x_1^{(q)}, \ldots, x_N^{(q)}) \qquad (9.11)$$

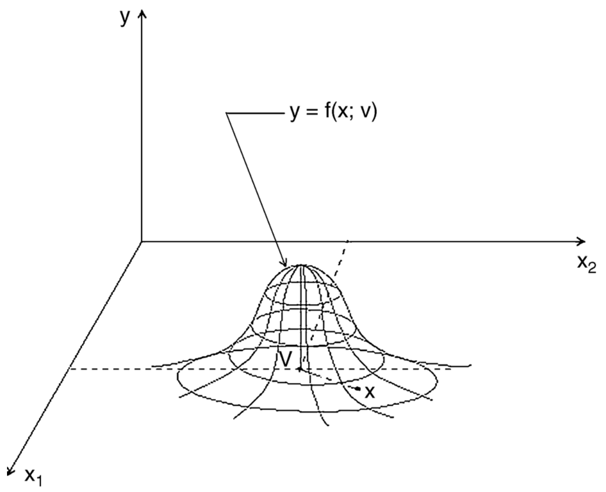

FIGURE 9.6 A radial basis function centered on *v*.

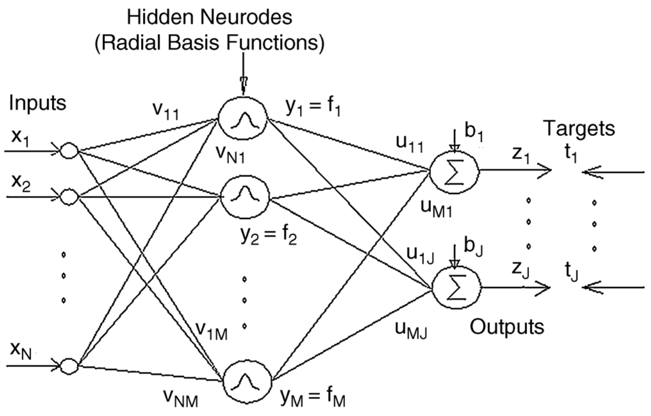

FIGURE 9.7 The radial basis function neural network architecture.

Each qth exemplar feature vector $\boldsymbol{x}^{(q)}$ corresponds to a hidden node as the center for a Gaussian (RBF). Each jth output node receives all of the values $\{y_{mj}\}_{m=1,M}$ from the M hidden-node Gaussians and takes a weighted average of them to obtain the final output value from the jth output node. All y_m values are less than or equal to 1, and all weights w_{mj} are less than or equal to 1, so the weighted sum is less than or equal to M. Thus, we divide the sum by M to keep the output less than or equal to 1 in magnitude.

Each actual output $z_j^{(q)}$ from the jth output node (for the input $\boldsymbol{x}^{(q)}$) is forced to match the target componnent label $t_j^{(q)}$ by adjusting the weights $\{w_{mj}\}$ (for all m). A bias b_j is added onto the sum and is also adjusted to help approximate the target value $t_j^{(q)}$, as shown in Figure 9.7.

All of the exemplar feature vectors are usually used as centers of RBFs. The output from the mth hidden node for any input vector $\boldsymbol{x}^{(q)}$ has the following form:

$$y_m^{(q)} = g_m(\boldsymbol{x}^{(q)}) = \exp[-(\boldsymbol{x}^{(q)} - \boldsymbol{x}^{(m)})^2/(2\sigma^2)] \tag{9.12}$$

The output value from each of the J nodes in the output layer for input $\boldsymbol{x}^{(q)}$ is

$$z_j^{(q)} = (1/M)\sum\nolimits_{m=1,M} w_{mj}\, y_m^{(q)} + b_j \tag{9.13}$$

where b_j is a bias to account for translation of the output values.

The training starts with an initial set of weights that is drawn randomly such that each weight is between –0.5 and 0.5. The basic idea is to adjust the weights $\{w_{mj}\}$ to minimize the *total sum-squared error* (TSSE) E between the actual outputs and the target output labels. E is defined over all exemplar feature vectors $\{\boldsymbol{x}^{(q)}: q = 1, \ldots, Q\}$ as inputs and over all J output values as:

$$E = \sum\nolimits_{q=1,Q}\sum\nolimits_{j=1,J}(t_j^{(q)} - z_j^{(q)})^2 \tag{9.14}$$

We could set the derivative $\partial E/\partial w_{mj}$ of E with respect to each weight equal to 0, and if we have the same number of equations and unknowns, we could solve the set of linear equations in the same number of unknowns. Usually these numbers are different, but even if they are not, the equations may be ill conditioned (two equations are approximately a multiple of each other), and the convergence may not be stable. We could compute a reduced basis of vectors for which the system is not ill conditioned, but this appears not to be as fruitful as the usual method of *steepest descent*.

Starting with a set of random weights $\{w_{mj}\}$ between –0.5 and 0.5 from a uniform random number generator, the convergence is quick. The iterative procedure is

$$w_{mj}^{(i+1)} = w_{mj}^{(i)} - \alpha(\partial E/\partial w_{mj}) \tag{9.15}$$

where the superscripts $i+1$ and i designate the iteration numbers. This is the familiar steepest descent method, where we start at an initial point and then move in the direction of steepest descent of E. Because E is a strictly convex function of the w_{mj}, it has a single minimum that is the global minimum.

The RBFNN Training Algorithm

Step 1. Read in all exemplar feature vectors in $\{\boldsymbol{x}^{(q)}: q = 1, \ldots, Q\}$ and their associated target vectors (code words or labels) in $\{\boldsymbol{t}^{(q)}: q = 1, \ldots, Q\}$, set iteration number $i = 0$.

Step 2. Compute the initial value $s = (1/2)[1/M]^{1/N}$ for the Gaussians.

Step 3. Input an integer value M for the number of exemplar feature vectors to use as centers (M may be equal to Q if Q is not too large).

Step 4. Select initial weights $\{w_{mj}\}$ and biases $\{b_j\}$ randomly between –0.5 and 0.5.

Step 5. Compute $\{y_m^{(q)}\}$ for all m for every input feature vector $\boldsymbol{x}^{(q)}$ (use Eq. (9.12)).

Step 6. Compute the outputs $\{z_j^{(q)}\}$ for all j and q (use Eq. (9.13)).

Step 7. Compute the TSSE value E_0 (use Eq. (9.14)).

Step 8. Iterate over the set of all M hidden nodes and all J output nodes:

```
for j = 1 to J do                                  //For all output nodes and for all
  for m = 1 to M do                                //hidden nodes, adjust weights
      $w_{mj}^{(i+1)} = (w_{mj}^{(i)} - \alpha(\partial E/\partial w_{mj})) = w_{mj}^{(i)} + (\alpha/M)\sum_{q=1,Q}(t_j^{(q)} - z_j^{(q)})(y_j^{(q)})$
  $b_j^{(i+1)} = b_j^{(i)} + (\gamma/M)\sum_{q=1,Q}(t_j^{(q)} - z_j^{(q)})$          //for (i+1)st iteration
  i = i + 1;                                       //Update iteration number
```

Step 9. Compute a new value for E (use Eq. (9.14)).

```
Step 10. If ($|E_0 - E| < \epsilon$) then stop;             //Stop if error change negligible
   else
      $E_0 = E$;                                       //or else save current error
      go to Step 8;                                    //and repeat adjustments
```

The *step sizes* α and γ are also called *learning rates*. These are numbers that usually start low (about 0.4). We then apply our *en route* method on α so that if $E < E_0$, then the step was a success and we increase the learning rate by $\alpha = 1.24\alpha$ to speed up the convergence, or else we decrease it by $\alpha = 0.96\alpha$. There are various strategies, but it is possible to increase α rapidly (a **greedy algorithm** that takes chances to converge very quickly) to several hundreds and then decrease it as needed as the minimum is approached.

9.3.4 Radial Basis Functional Link Nets

If we start with the RBFNN and add extra lines from the input nodes to the output nodes and also weight these with weights $\{u_{nj}\}$, then we have two sets of values to average at the output nodes: (1) the x_n from the input nodes weighted by u_{nj}, and (2) the y_m from the hidden nodes weighted by w_{mj}. Thus, the outputs

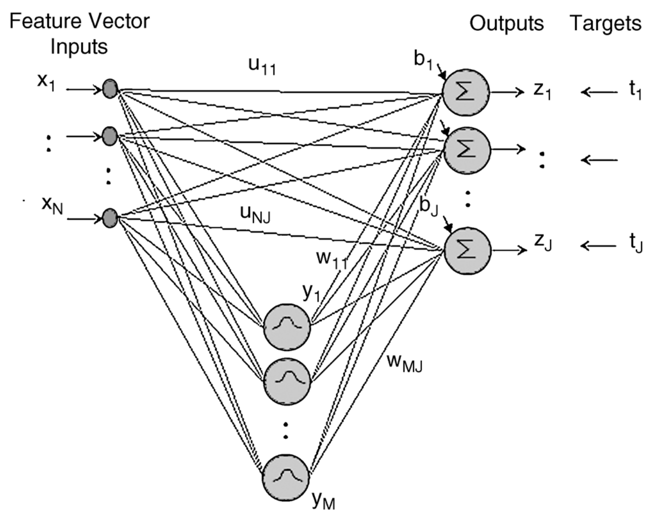

FIGURE 9.8 The radial basis functional link net architecture.

from the nodes at the output layer now become

$$z_j = [1/(M+N)]\left\{\textstyle\sum_{m=1,M} w_{mj}\, y_m^{(q)} + \sum_{n=1,J} u_{nj}\, x_n^{(q)} + b_j\right\} \tag{9.16}$$

We call this network a **radial basis functional link net** (RBFLN) [Looney, 2001a] (Looney calls it a *random vector quantization functional link net* (RBQFLN) [1997]). The name originated with the functional link net of Pao et al. [1994]. The RBFLN is more powerful than the RBFNN because it includes the linear combinations of the inputs as well as the nonlinear part from the hidden nodes and the biases. In this case, we also adjust the new weights $\{u_{nj}\}$, in addition to adjusting the previous weights $\{w_{mj}\}$ and biases $\{b_j\}$, all by steepest descent iteration.

Figure 9.8 shows the radial basis functional link-net architecture. These networks perform better than RBFNNs, which in turn perform better than the so-called *multiple-layered perceptrons* (MLPs), also known as *feed forward neural networks*, with the so-called *back-propagation algorithm*, which is also steepest descent [Looney, 1997], but that is problematic due to multiple local minima.

To adapt the RBFNN to that of the RBFLN we must also randomly initialize the second set of weights $\{u_{nj}\}$ in addition to $\{w_{mj}\}$ in Step 4 above and train them on the second set of weights as well as the first. The iterative Step 8 in the previous RBFN algorithm changes as given below.

The RBFLN Algorithm (only Step 8 need be shown here)

Step 8. Iterate weight and bias adjustments over the set of all output, hidden and input nodes:

```
for j = 1 to J do                                   //For each outut node: for each
for m = 1 to M do                                   //hidden node, update weights
        $w_{mj}^{i+1} = (w_{mj}^{(i)} - \alpha(\partial E/\partial w_{mj})) = [a/(M+N)]\sum_{q=1,Q}(t_j^{(q)} - z_j^{(q)})(y_m^{(q)})$
for n = 1 to N do                                   //and for each input node, update weights
        $u_{nj}^{i+1} = (u_{nj}^{(i)} - \beta(\partial E/\partial u_{nj})) = [\beta/(M+N)\sum_{q=1,Q}(t_j^{(q)} - z_j^{(q)})(x_n^{(q)})$
$b_j = b_j + (\gamma/M)\sum_{q=1,Q}(t_j^{(q)} - z_j^{(q)})$          //Update bias at each output node
```

The advantage of using the radial basis functional link net (RBFLN) over the RBFNN is that it adds a linear input–output functional component to the outputs so that any linear relationships do not need to be approximated by the nonlinear part coming through the hidden nodes. Looney has shown [2001a] over many simulation runs that it learns with significantly fewer iterations and learns better (fewer mistakes). There appears to be no good reason to use the RBFNN instead of the RBFLN.

9.3.5 A Simplified Approach to RBFNNs and RBFLNs

A promising modification to the RBFNN and the RBFLN to make them more efficient is the reduction in the number of Gaussians. This also prevents extraneous error from building up in the summing of the Gaussians at the output nodes. It involves the thinning of the exemplar vectors in each class, especially for classes with large numbers of exemplars.

For each Class k in turn, we examine the set of all (labeled) exemplars $\{\boldsymbol{x}^{(q)}: q = 1, \ldots, Q\}$ to find those that belong to Class k via $c[q] = k$. We save these in a new data structure of vectors and then perform a search with each one to see if there are any other exemplars in this class that are too close, according to a threshold T_k for Class k. We take T_k to be a constant α times the average distance between feature vectors in that class, where $0.4 < \alpha < 0.8$ (a higher value yields fewer exemplars in Class k). We eliminate any exemplar vectors that are too close to the current one and reindex. Then we select the next exemplar remaining in the class and search for exemplars in the class that are too close to it. We continue this process until every exemplar vector in this class has been checked (this is the same thinning process that we used in the improved k-means algorithm).

After such processing, we use the remaining exemplar vectors for each class as centers for Gaussians as before. At this point we have a smaller subset of Gaussians that is sparser but distributed over the classes so that the processing is more efficient and the extraneous errors are smaller. Figure 9.9 shows a thinned set of five mixed Gaussians that cover a class of exemplar vectors (the class need not be a circular region and usually is not).

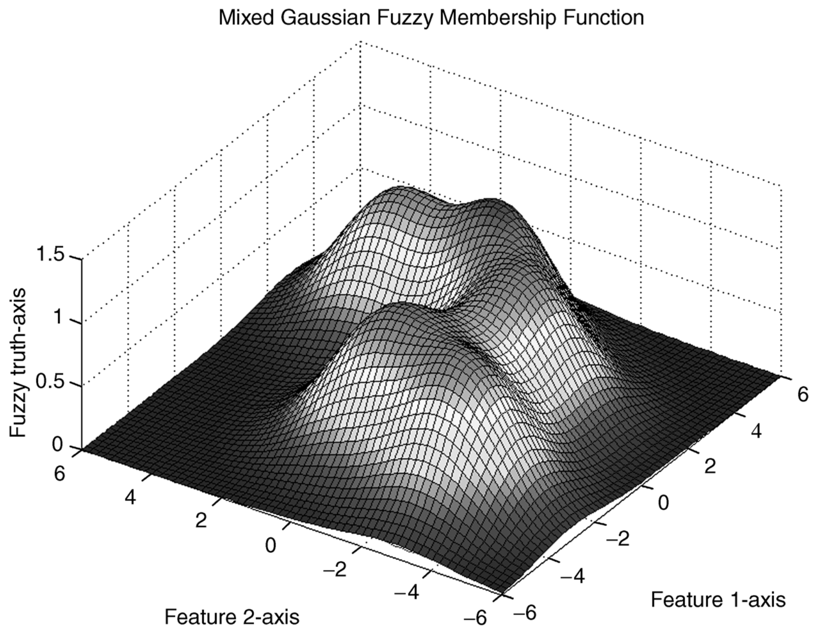

FIGURE 9.9 Thinned multiple Gaussians covering a cluster.

9.3.6 Using Ellipsoidal Basis Functions

In the algorithms given above we have used radial (circular) Gaussians as the radial basis functions. It is also possible to use *ellipsoidal basis functions* that have been examined in recent research by Abe and Thawonmas [1997] and Looney [2001a; 2002b]. To use these, we must first perform classification to get labeled feature vectors, separate them by class to obtain a data structure for each class of these exemplar vectors, and compute the respective covariance and inverse covariance matrices C_k and C_k^{-1} for each kth class. Then we can use the full multivariate Gaussian function for the resulting ellipsoidal (nonradial) basis functions, given by:

$$g_k(\boldsymbol{x}) = \exp\{-(1/2)(\boldsymbol{x} - \boldsymbol{z}^{(k)})^t C^{-1}(\boldsymbol{x} - \boldsymbol{z}^{(k)})\} \tag{9.17}$$

where k designates the kth class and $\boldsymbol{z}^{(k)}$ is the prototype (center) for that class.

Classes often do not fit inside circles, so we use the data in each class to shape an ellipsoid to contain the class exemplar vectors. The *generalized Gaussian* is defined by Eq. (9.17) above. The lines of equal functional value of this Gaussian form not circles, but ellipses instead. We can use ellipsoidal basis functions for training RBFNNs and RBFLNs, in which case they become, respectively, **ellipsoidal basis function NNs** (EBFNNs) and **ellipsoidal basis functional-link nets** (EBFLNs). In this case we can use the mean, median, the α-trimmed mean [Bednar and Watt, 1984], or the weighted fuzzy value as the single prototype of each class. We can multiply the covariance matrix C by a factor $\theta > 1.0$ (which multiplies the covariance matrix C^{-1} by $1/\theta$) to expand the ellipses slightly for better results ($\theta = 1.2$ is an empirical value that improves performance for ellipsoidal Gaussians).

9.4 Fuzzy Classifiers

9.4.1 Using a Fuzzy Ellipsoidal Classifier

Given the exemplar feature vectors $\{\boldsymbol{x}^{(q)}: q = 1, \ldots, Q\}$, we do the following steps to design a **fuzzy classifier** [Looney, 2001b] (see also Maturino-Lozoya et al. [2000]), which is actually a recognizer that is similar to a fuzzy neural network. It uses a fuzzy-set membership function for each class to provide the fuzzy truth that an input feature vector belongs to that class. The exemplar feature vectors must first be clustered and labeled. Here we use a **fuzzy ellipsoidal classifier,** so that this classifier is similar to the EBFNN, except that we have no weights to train.

A Fuzzy Ellipsoidal Classifier

Step 1. Obtain the Q-labeled exemplar feature vectors and separate the K classes of vectors.

Step 2. For each class, compute the mean, α-trimmed mean, median, or weighted fuzzy average, $\boldsymbol{c}^{(k)}$.

Step 3. For each Class k compute the covariance matrix C_k and its inverse C_k^{-1}.

Step 4. For each Class k define the ellipsoidal Gaussian fuzzy-set membership function:

$$g_k(\boldsymbol{x}) = \exp\{-[1/(2\theta)](\boldsymbol{x} - \boldsymbol{z}^{(k)})^t C^{-1}(\boldsymbol{x} - \boldsymbol{z}^{(k)})\} \tag{9.18}$$

where $\theta > 1.0$ is discussed above and $\boldsymbol{z}^{(k)}$ is the center of Class k.

Step 5. Implement the following algorithm for online processing of the input vector stream $\{\boldsymbol{x}\}$

```
Repeat                              //Iterate over input vectors
    read $\boldsymbol{x}$                          //Read next feature vector to recognize
    for k = 1 to K do               //For each Class k compute the
        f[k] = $g_k(\boldsymbol{x})$;               //fuzzy membership function on $\boldsymbol{x}$
```

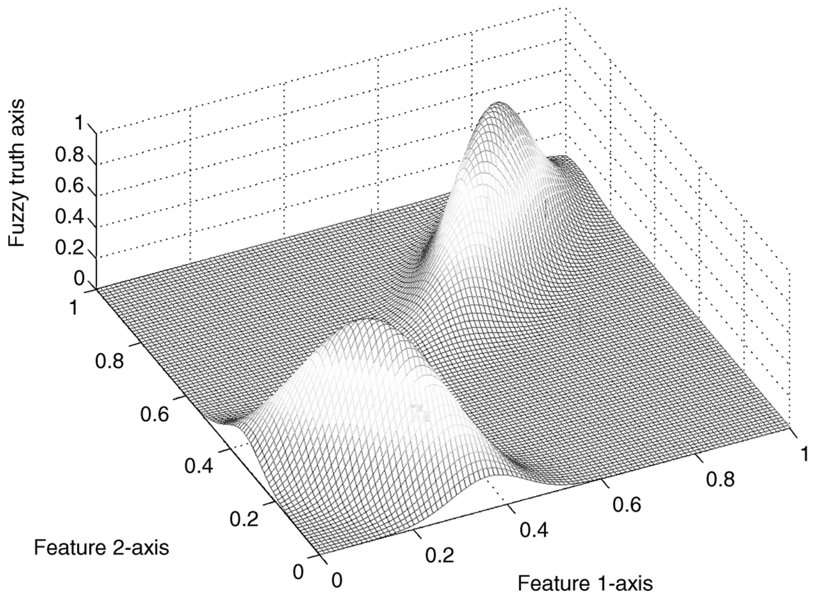

FIGURE 9.10 Ellipsoidal Gaussians covering two classes.

```
        fmax = 0.0;                  //Find the maximum fuzzy membership value
        for k = 1 to K do            //over all K classes and record
            if (f[k] > fmax) then
                fmax = f[k];         //the maximum value and the
                kmax = k;            //index for the maximum value
        output(fmax, kmax);          //Output the index kmax of winner
    until process is stopped;
```

Figure 9.10 shows two ellipsoidal Gaussians that cover two classes in the plane. These classes are well separated on synthetic data for the purpose of illustration, but classes may be closer together on real-world data. The maximum value *f[kmax]* of the fuzzy-set membership functions determines the winner, which is the Class *kmax* that $\boldsymbol{x}$ is recognized as belonging to.

9.4.2 Computing the Covariance Matrix

The covariance matrix must be computed for each class. We do this with the following algorithm, where $c[q] = k$ means that feature vector $\boldsymbol{x}^{(q)}$ belongs to the kth class, $\{cv[i][j]\colon 1 \le i, j \le N\}$ is the computed covariance matrix, and $c[n][k]$ is the nth component of the center for the kth class.

The Covariance Matrix Algorithm

```
Step 1: for k = 1 to K do                          //For each Class k
            for n = 1 to N do                      //and each component
                read x[n][k] for Class k;          //input class exemplars
            for n = 1 to N do                      //Similarly for the prototypes
                read prototype vector z[n][k] for Class k;
```

```
Step 2: for k = 1 to K do                                      //For each class, process to get C
           for $n_1$ = 1 to N do                                 //For each first component
               for $n_2$ = $n_1$ to N do                          //and second component
                   sum = 0.0;                                  //initialize sum, check each feature
                   for q = 1 to Q do                           //vector to determine if it belongs to Class k
                       if (c[q] = = k) then                    //and if so, process it in covariance value
                           sum = sum + (x[$n_1$][q] – z[$n_1$][k])*(x[$n_2$][q] – z[$n_2$][k]);
                   if (count [k] > 1) then                     //Average if more than 1 vector
                           cv[$n_1$][$n_2$][k] = sum/(count[k] – 1);
                   else                                        //else covariance is 0
                           cv[$n_1$][$n_2$][k] = 0.0;
                   if ($n_1$! = $n_2$) then                      //If not diagonal element then
                           cv[$n_2$][$n_1$] = cv[$n_1$][$n_2$];      //fill in matrix by symmetry
```

9.4.3 Computing the Inverse Covariance Matrix

To compute the inverse C^{-1} of the covariance matrix C of the kth class, we form an extended matrix by putting C and the $N \times N$ identity matrix I together in $[C \mid I]$, which is $N \times (2N)$ in size. We first do the upper triangularization on C to zero out all entries in C below the diagonal and then perform lower triangularization to zero out everything above the diagonal of C. Then, we normalize the diagonal elements so that the part that was C becomes the identity matrix I. For every operation that we do to C, we do to I also, so that by transforming C into I we also transform I into C^{-1} at the same time; that is, $[C \mid I]$ is transformed to $[I \mid C^{-1}]$. In the algorithm below, we denote the extended matrix $[C \mid I]$ for the kth class by $cx[n][col][k]$, where n is the row and col is the column.

The Inverse Covariance Matrix Algorithm

```
Step 1: Upper triangularization of extended matrix
        for k = 1 to K do                                      //For each kth class
           for col = 0 to N do                                 //Eliminate below each column
               for $n_1$ = col to N – 1 do                       //row entries (make zero)
                   if (cx[$n_1$ +1][col][k] ! = 0) then          //If element not 0 pivot on it,
                       t = cx[$n_1$ +1][col][k]/cx[col][col][k]; //get elimination factor
                       for $n_2$ = col to 2N do                  //and apply to column elements
                           cx[$n_1$ + 1][$n_2$][k] = cx[$n_1$ + 1][$n_2$[k] – t*cx[col][$n_2$][k]
Step 2: Lower triangularization of extended matrix
        for k = 1 to K do                                      //For each class
           for n = 1 to N do                                   //go through extended matrix
               col = N – n;                                    //and get pivot element
               for m = 0 to col do                                   //for elimination
                   n_1 = col – m – 1;
                   if (cx[n_1][col][k] ! = 0.0 then                    //If element not 0, then use as pivot
                       t = cx[n_1][col][k]/cx[col][col][k];            //Compute elimination factor
                       for n_2 = col to 2N do                          //and apply to column elements
                           cx[n_1][n_2][k] = cx[n_1][n_2][k] – t*cx[col][b_2][k];
```

Step 3: Normalize diagonal elements in extended matrix

```
    for k = 1 to K do                                         //For each class
      for n_1 = 1 to N do                                     //go through rows and divide
        t = 1.0/cx[n_1][n_1][k];                              //by diagonal element
        for n_2 = n_1 to 2N do                                //across all columns from diagonal
          cx[n_1][n_2][k] = t*cx[n_1][n_2][k];                //element to the right
```

Step 4: Retrieve inverse covariance matrix from extended matrix

```
    for k = 1 to K do                                         //For each class retrieve inverse
      for n_1 = 1 to N do                                     //from [I | C^{-1}], so get all rows
        for n_2 = N + 1 to 2N do                              //and all columns on right
          ci[n_1][n_2][k] = cx[n_1][n_2 +N][k];               //This N × N matrix is C^{-1}
```

We can check C to see if it is invertible (nonsingular) before we try to invert it. C is symmetrical with nonnegative diagonal elements. A practical check can be made on the upper triangularized matrix $C^{[T]}$ by testing the diagonal elements to ensure that none are approximately 0 (recall that the determinant of a triangular matrix is the product of the diagonal elements). If all exemplar feature vectors in a class have the same value for a given feature, then a diagonal element (a component variance) will be zero and C (and its upper triangularization) will be singular. This should never occur on real world data where the features have been selected for their separation properties.

9.5 An Application: Edge Recognition in Images

9.5.1 Image Edge Detection

Edges are defined as locations in an image where there is a significant variation in the gray level or intensity of color of pixels in some direction across a small number of pixels [Efford, 2000]. They are one of the most important visual clues for interpreting images [Gose et al., 1996]. The process of **edge detection** reduces an image to show only its edges, which appear as the outlines of objects within the image that can be used in subsequent image analysis operations for feature detection and object recognition. Although there are many different methods for edge detection, such as Sobel 3×3 filtering, Prewit 3×3 filtering, Laplacian of Gaussian filtering, moment-based operators, the Shen and Castan operator, and the Canny and Deriche operator, some common problems of these methods are a large volume of computation and too much sensitivity to noise and anisotropy.

Russo [1992, 1993] and Russo and Ramponi [1992] designed fuzzy rules for edge detection. Such rules can smooth while sharpening edges, but this method requires a rather large rule set [Looney, 2000]. Here we describe a special fuzzy classifier for edge detection that does not require training [Liang and Looney, 2001]. Our fuzzy classifier uses the two classes of *edge* and *background*. Its advantages are easy modeling, efficient computation, low sensitivity to noise, and isotropy (it detects edges in all directions).

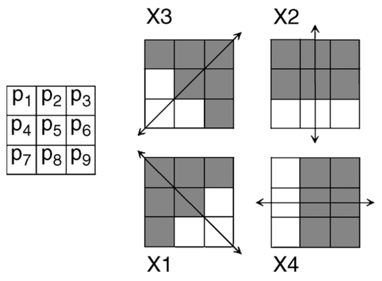

FIGURE 9.11 The edge classes.

9.5.2 Fuzzy Edge Detection

Figure 9.11 shows the 3 × 3 neighborhood of the center pixel p_5 and also the four directions of gray-level change. The magnitudes of the gray-level changes in these directions are defined in the horizontal, vertical, and two diagonal directions and are designated by, respectively, X1, X2, X3, X4. The four features are calculated by

$$X1 = |p_1 - p_5| + |p_9 - p_5|, \quad X2 = |p_2 - p_5| + |p_8 - p_5| \tag{9.19a,b}$$

$$X3 = |p_3 - p_5| + |p_7 - p_5|, \quad X4 = |p_4 - p_5| + |p_6 - p_5| \tag{9.19c,d}$$

For each image pixel (not on the outer boundary of the image), we compose the four-dimensional feature vector $\boldsymbol{x} = (X1, X2, X3, X4)$ that contains the gray-level difference magnitudes in the four directions of its 3 × 3 neighborhoods. X1 is diagonal (upper left to lower right); X2 is vertical (downward along the center column); X3 is diagonal (upper right to lower left); and X4 is horizontal (rightward along the center row).

9.5.3 Pixel Classes and Their Feature Vectors

Four classes of edges and a fifth background class are differentiated in our **competitive fuzzy classifier**. Four edge situations are used for each class, which are those shown in Figure 9.11, their opposites determined by 180% rotation and those with the dark and light pixels reversed. The five prototypical feature vectors are designated by $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_4,$ and each represents four neighborhood situations.

We use the linguistic variable *Low* to substitute for 0, and *High* to substitute for 255 in the directional difference magnitudes (see Figure 9.11). Letting *L* denote *Low* and *H* denote *High*, the feature vectors become:

Background pixel, Class 0: $\boldsymbol{x}_0 = \{L, L, L, L\}$

Edge pixel, Class 1: $\boldsymbol{x}_1 = \{L, H, H, H\}$ Edge pixel, Class 2: $\boldsymbol{x}_2 = \{H, L, H, H\}$

Edge pixel, Class 3: $\boldsymbol{x}_3 = \{H, H, L, H\}$ Edge pixel, Class 4: $\boldsymbol{x}_4 = \{H, H, H, L\}$

In practice, the values of *Low* and *High* can be defined by the user for each particular image to achieve a desirable result, for example, $L = 5$ and $H = 20$ gray levels. With these values defined, every interior pixel of the input image can be classified under one of the above classes by its feature vector of directional difference magnitudes on its 3 × 3 neighborhood.

9.5.4 The Fuzzy Classifier Architecture

Our *competitive fuzzy classifier* is made up of our basic fuzzy classifier and some competitive rules that allow a competition between pixels for designation as an edge (see Chen and Chi [1998] and Chung and Lee [1994] for other competitions). Our fuzzy classifier classifies each pixel in an image as a white background pixel, or else one of the four classes of edge pixels shown above, by providing fuzzy truths

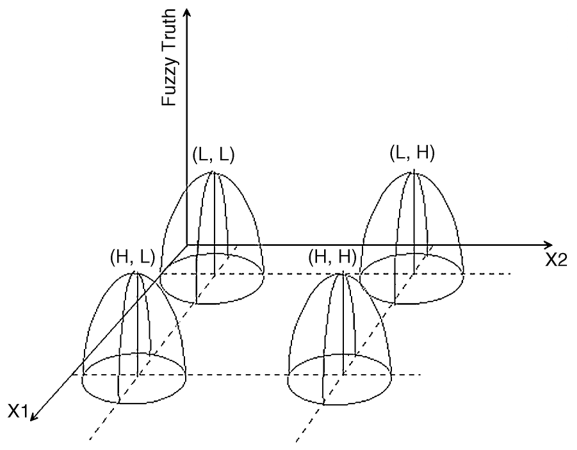

FIGURE 9.12 A two-dimensional depiction of the fuzzy edge classifier.

of those classes based on the feature vector of that pixel. Then our competitive rules [Liang and Looney, 2001] are applied according to the output of the original fuzzy classifier. Only the pixels that are classified as edge (directional) pixels and that subsequently win in the local competition are mapped to black edges in the new output image. This creates a black line drawing on a white background. If there are adjacent edge pixels in the same direction, only the one with the greatest fuzzy value is the winner.

On the four-dimensional feature space, we define the **fuzzy membership functions** for the five classes with *extended Epanechnikov* [Looney, 2001b] functions defined by:

$$\text{Background Class 0: } f(\boldsymbol{x}) = \max\{0, 1 - \|\boldsymbol{x} - \boldsymbol{x}_0\|^2/\beta^2\} \tag{9.20a}$$

$$\text{Edge Class 1: } f(\boldsymbol{x}) = \max\{0, 1 - \|\boldsymbol{x} - \boldsymbol{x}_1\|^2/\beta^2\} \tag{9.20b}$$

$$\text{Edge Class 2: } f(\boldsymbol{x}) = \max\{0, 1 - \|\boldsymbol{x} - \boldsymbol{x}_2\|^2/\beta^2\} \tag{9.20c}$$

$$\text{Edge Class 3: } f(\boldsymbol{x}) = \max\{0, 1 - \|\boldsymbol{x} - \boldsymbol{x}_3\|^2/\beta^2\} \tag{9.20d}$$

$$\text{Edge Class 4: } f(\boldsymbol{x}) = \max\{0, 1 - \|\boldsymbol{x} - \boldsymbol{x}_4\|^2/\beta^2\} \tag{9.20e}$$

where $\boldsymbol{x}$ is any input feature vector for a pixel and $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_4$ were defined above. Thus, the quality of the edge detection, as measured by the fuzzy truth of its memberships in the fuzzy classes, depends on the parameters L, H, and β, and thus on the particular image.

Figure 9.12 provides a two-dimensional portrayal of two features vs. fuzzy truths for easy visualization. The upside-down cups in the figure are the extended Epanechnikov functions [Looney, 2001b] that are shown here with small diameters for clarity (in applications they overlap). Each input feature vector $\boldsymbol{x}$ falls into one or more of these fuzzy-set membership functions. Given any input feature vector $\boldsymbol{x}$, the maximum fuzzy truth-value of the three shown fuzzy-set membership functions evaluated on X determines the class of the pixel.

9.5.5 Competitive Edge Rules

Before an edge pixel is changed to either white or black in the output image, a competition with its neighbor edge pixels is conducted. For the edge pixel neighbors that belong to the same class of edge, only the one with the largest feature (difference magnitude in the direction associated with that class) is considered an edge and turned to black. Thus, the edges in the output image will be thin instead of thick. Rules for the competition are given below.

IF (white class wins) THEN (change to white).

IF (edge Class 1 wins) THEN compete X3 with neighbor pixels in direction 3.

IF (win) THEN (change to black) ELSE (change to white).

IF (edge Class 2 wins) THEN compete X4 with neighbor pixels in direction 4.

IF (win) THEN (change to black) ELSE (change to white).

IF (edge Class 3 wins) THEN compete X1 with neighbor pixels in direction 1.

IF (win) THEN (change to black) ELSE (change to white).

IF (edge Class 4 wins) THEN compete X2 with neighbor pixels in direction 2.

IF (win) THEN (change to black) ELSE (change to white).

9.5.6 The Algorithm

Figure 9.13 is a flowchart that shows the algorithm in detail for operating on PGM files of grayscale images, which could also be intensity data from color images. The third block from the top processes each pixel not on the image boundary to compute its fuzzy truth of membership in each of the five

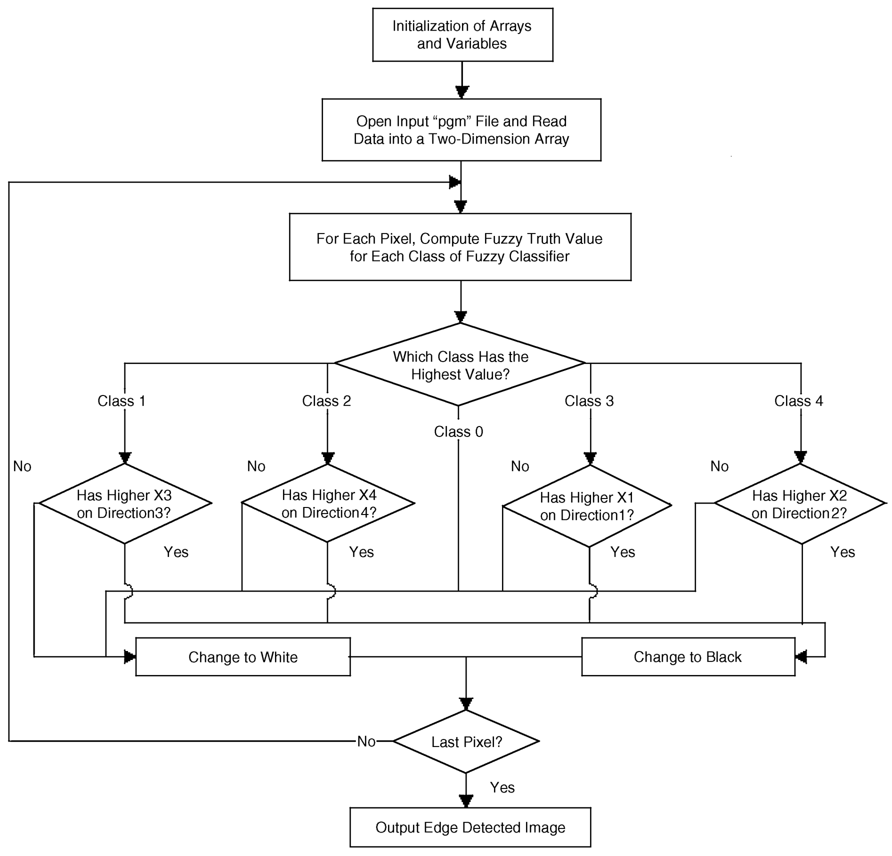
Initialization of Arrays and Variables
Open Input "pgm" File and Read Data into a Two-Dimension Array
For Each Pixel, Compute Fuzzy Truth Value for Each Class of Fuzzy Classifier
Which Class Has the Highest Value?
Class 1
Class 2
Class 0
Class 3
Class 4
No
Has Higher X3 on Direction3?
Yes
No
Has Higher X4 on Direction4?
Yes
No
Has Higher X1 on Direction1?
Yes
No
Has Higher X2 on Direction2?
Yes
Change to White
Change to Black
No
Last Pixel?
Yes
Output Edge Detected Image

FIGURE 9.13 The algorithm flowchart.

classes. The winning class determines the rule to be used in the competitive comparisons to determine whether a pixel is converted to white (background) or black (edge).

9.5.7 Edge Detection Results

All of our results are obtained by using a 3 × 3 neighborhood of the center pixel and the fuzzy-set membership functions and rules established above. The threshold parameters (L and H) are adjusted to achieve good results.

Figure 9.14 shows the original image *building.pgm*. The results of using the public-domain software *XView* that is included with the *Linux* operating system on the image are shown in Figure 9.15. The results of using the Canny edge detector with respective low and high edge sensitivity are shown in Figures 9.16 and 9.17. The competitive fuzzy-edge detection results for the respective high and low sensitivity to edges are shown in Figures 9.18 and 9.19, and they were obtained in less than 1/20 of the computing time required for the Canny edge detector.

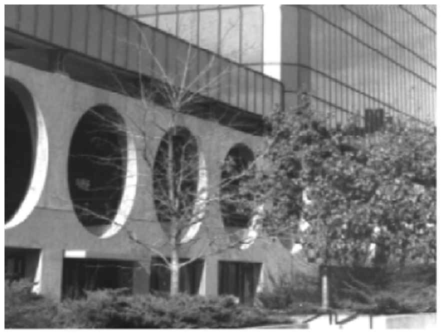
FIGURE 9.14 The original *building.pgm* image.

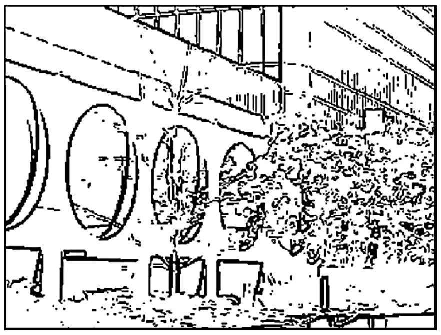
FIGURE 9.15 The *XView* edges.

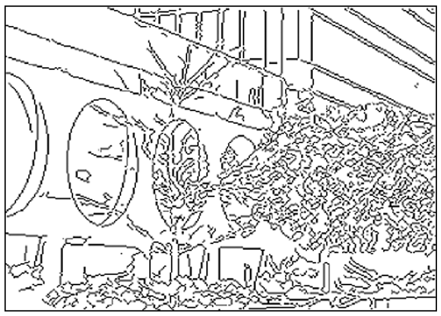
FIGURE 9.16 The Canny edges, low sensitivity.

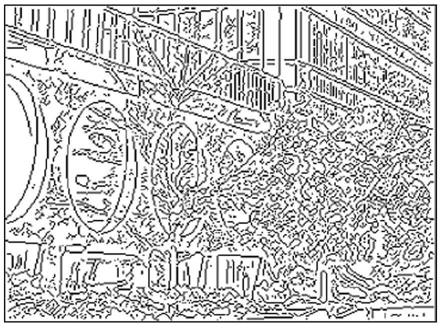

FIGURE 9.17 The Canny edges, high sensitivity.

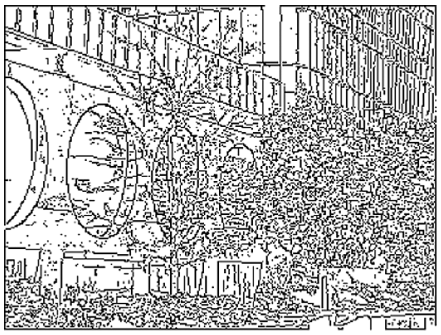

FIGURE 9.18 Competitive fuzzy edges, higher sensitivity.

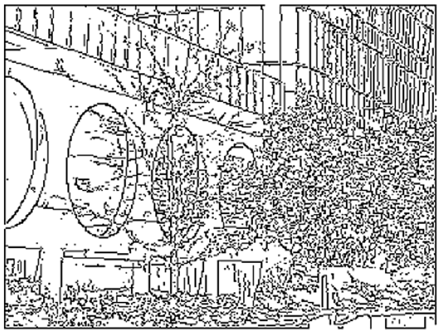

FIGURE 9.19 Competitive fuzzy edges, lower sensitivity.

9.6 Summary

We have discussed classification and recognition, although the literature at times uses them interchangeably. For classification, the problems with the *k*-means algorithm for clustering were listed and two improved algorithms were provided, of which the second was our weighted fuzzy *k*-means algorithm. For recognition, we have provided the following powerful algorithms: (1) probabilistic neural networks; (2) fuzzy neural networks; (3) radial basis function neural networks; (4) radial basis functional link nets; (5) ellipsoidal-basis function neural networks; (6) ellipsoidal radial basis functional link nets; and (7) fuzzy ellipsoidal classifiers.

For the networks that use Gaussian radial basis functions, we have suggested a thinning algorithm that can be used to reduce the number of mixed Gaussians to provide efficiency and also to alleviate the build-up of error from extraneous noise. By choosing a combination of the clustering and recognition algorithms presented here, one can attack most classification and pattern recognition problems successfully. However, it is possible to modify any of these for special purposes.

As an application, we have given an example of fuzzy edge detection and an improvement called *competitive fuzzy edge recognition* for yielding thinner and cleaner lines. This latter method can be combined with some preprocessing, such as frequency filtering, to remove the highest frequencies of noise or despeckling. We have put the neighborhood directional difference magnitudes into the competitive fuzzy classifier to classify a pixel as being of type *edge* or *background* and then thinned the edges. The result is a line drawing of thin black lines on a white background, whereas the usual methods yield thick lines, or, in the case of the Canny edge detector, noise is converted to a plethora of mostly useless edge traces.

Defining Terms

α-trimmed mean: A representative value for a set of values determined by omitting the α smallest and the α largest values and averaging the remainder.

artificial intelligence: Any machine process that emulates human reasoning or decision making by means of deduction/abduction, computational methods (statistical/mathematical), fuzzy methods, genetic algorithms, or other means.

center: The typical (prototypical) representative feature vector for a class of feature vectors that represent objects in a population of such objects.

classes: Groups of objects such that those within a group are similar according to some criteria and those in different groups are dissimilar according to the same criteria.

classification: A set of classes determined for a population; a research area of study; a process of determining a set of classes (see *classify*).

classify: The process of partitioning the objects of a population into groups (*classes*) that are similar within each group and dissimilar between groups.

clustering: The process of partitioning a population of objects into groups to form classes, a type of classification.

clustering validity measure: A measure of a clustering (classification) to determine how good the clustering is, used in a relative sense to compare multiple clusterings.

clusters: Groups of objects that form a partition of a population of objects.

compact: The property of a class of feature vectors having a small mean square error (variance) relative to the distances to the centers of other classes.

competitive fuzzy classifier: A fuzzy classifier that implements a competition between objects that are classified by a fuzzy classifier and there is a single winner of the competition for some purpose (such as determining which of neighboring edge pixels is the edge).

edge detection: A process of finding (detecting or recognizing) the pixels in an image that are edges.

ellipsoidal basis function: A multivariate (ellipsoidal) Gaussian function centered on a vector.

ellipsoidal basis functional link net: A radial basis functional link net where one or more radial basis functions are replaced by ellipsoidal basis functions.

ellipsoidal basis function neural network: A neural network similar to a radial basis function neural network except that the ellipsoidal basis functions replace one or more radial basis functions.

exemplar: A feature vector that is labeled with a particular class codeword (it contains extra components for the label).

feature: A property of a population of objects that can be observed (measured, detected) as a real number value for any selected object.

feature vector: A vector of observed feature values for a fixed set of features for a given population of objects.

fuzzy classifier: A system such that a feature vector is input and the outputs are the fuzzy truths of the memberships of the feature vector in the various classes, a fuzzy recognizer.

fuzzy ellipsoidal classifier: Fuzzy classifier where one or more of the fuzzy-set membership functions are ellipsoidal.

fuzzy neural network: A neural network with *N* input nodes corresponding to the components of a feature vector such that either the inputs are feature-wise fuzzy truths, or such that the final outputs represent the respective fuzzy truths of memberships in the various classes (or in some cases, both fuzzy truth inputs and outputs).

fuzzy-set membership function: A function on a real or vector domain whose values are between 0 and 1 and whose value represents the fuzzy truth of a property of the argument.

hidden layer: A set of nodes in a network that neither receive inputs from the outside nor send output values to the outside, nodes that connect only to other nodes in the network.

identification: The process of recognizing a unique individual from a population of objects.

input layer: A set of nodes in a network each of which receives the input of a component of some data structure (such as a vector) from a source external to the network.

***k*-means:** An algorithm that iterates: (1) the assignment of feature vectors to class centers to form new classes and (2) the determination of new class centers by averaging the vectors in each class, where the seeds (initial centers) are randomly selected feature vectors.

label: An integer or code word that represents the class to which the feature vector for an object belongs, assigned by humans or assigned by a clustering process.

maximum *a posteriori*: Also designated as MAP, the maximum of the probabilities of membership of an input feature vector in the various classes that is computed from the input feature vector and the prior probability distributions (in this case, mixed Gaussians).

output layer: A set of nodes in a network that receive input values from nodes within the network and send output values to destinations or sinks outside of the network.

pattern recognition: The process of recognizing an object as belonging to a particular class of the population by working on a set of measurement values for the object features (attributes).

probabilistic neural network: A network that accepts feature vectors as inputs and outputs the probabilities of belonging to particular classes.

prototype: A center or other representative vector for a class or cluster that represents the class.

radial basis function: A circular Gaussian centered on a vector; that is, a function whose value is the same for all vectors of equal distance to the center.

radial basis functional link net: A radial basis function neural network that has extra lines from the input nodes to the output nodes and an extra set of weights on these lines (thus a linear part is added to the model for efficiency and accuracy).

radial basis function neural network: A neural network with an input, a hidden, and an output layer of nodes where: (1) each input node accepts a component of a feature vector; (2) each hidden node receives all input components and outputs the value of the Gaussian radial basis function centered on its associated exemplar feature vector; and (3) the output nodes sum the weighted radial basis function values from the hidden nodes (the training on exemplars adjusts the weights by steepest descent until each input feature vector yields a correct approximate output target).

recognition: The same process as pattern recognition.

recognizer: An online process that performs recognition of feature vectors from a particular population on which it has been trained.

seed: An initially given center (prototype) on which to form a class as the first step in a classification (clustering) process.

self-organization: A system that interacts with an environment and adjusts itself in some fashion to perform one or more tasks more optimally relative to the interaction; a form of machine learning.

supervised learning: A process whereby a system is trained on examples to perform a task, for example, recognition.

target: The desired output number or vector that the actual computed outputs are to approach (approximately), usually by iterations.

training: A process of presenting examples and the correct responses to a system until the system parameters are adjusted to put out the correct responses to inputs.

unsupervised learning: A process whereby a local system interacts with an external system and processes the response data to adjust its parameters to more optimally interact with the system for some purpose; a form of machine learning (see *self-organization*).

weighted fuzzy average: The weighted average of a set of values obtained by an iterative process of taking weighted averages of N values, where the weights are initially $1/N$ and are computed as the values of a radial fuzzy set membership function centered on the current average (done component-wise for vectors), designated as *WFA*.

well separated: A property of a set of clusters (classes) whereby the centers of the clusters are located relatively far apart when compared to their variances.

Xie-Bene clustering validity measure: A measure computed for a particular clustering of a set of feature vectors that takes the sum of the cluster variances and divides by the minimum distance between cluster centers.

References

Abe, S. and Thawonmas, R., A fuzzy classifier with ellipsoidal regions, *IEEE Trans. Fuzzy Sys.*, 6(2), 358–368, 1997.

Anagnostopoulos, C., Anagnostopoulos, J., Vergados, D., Kayafas, E., Loumos, V., and Stassinopoulos, G., A neural network and fuzzy logic system for face detection on RGB images, *Proc. ISCA Int. Conf. Computers Their Applications*, 2001, pp. 233–236.

Bednar, J. B. and Watt, T. L., Alpha-trimmed means and their relation to median filters, *IEEE Trans. Acoustics*, Speech and Signal Processing, 32(1), 145–153, 1984.

Bezdek, J. C., Fuzzy Mathematics in Pattern Classification, Ph.D. thesis, Center for Applied Mathematics, Cornell University, Ithaca, NY, 1973.

Chen, K. and Chi, H., A method of combining multiple probabilistic classifiers through soft competition on different feature sets, *Neurocomputing*, 20, 227–252, 1998.

Chen, M. S. and Wang, S. W., Fuzzy clustering analysis for optimizing fuzzy membership functions, *Fuzzy Sets Systems*, 103, 239–254, 1999.

Chiu, S. L., Fuzzy model identification based on cluster estimation, *J. Intelligent Fuzzy Sys.*, 2, 3, 1994.

Chung, F. L. and Lee, T., Fuzzy competive learning, *Neural Networks*, 7(3), 539–551, 1994.

Dubes, R. C. and Jain, A. K., *Algorithms for Clustering Data*, Prentice-Hall, Englewood Cliffs, NJ, 1998.

Efford, N., *Digital Image Processing*, Addison-Wesley, Reading, MA, 2000.

Forgy, E., Cluster analysis of multivariate data: efficiency versus interpretability of classifications, *Biometrics*, 21, 768–776, 1965.

Gose, E., Johnsonbaug, R., and Jost, S., *Pattern Recognition and Image Analysis*, Prentice-Hall, Upper Saddle River, NJ, 1996.

Kaufman, L. and Rousseeuw, P., *Finding Groups in Data: an Introduction to Cluster Analysis*, John Wiley & Sons, NY, 1990.

Liang, L. and Looney, C., Competitive Fuzzy Edge Detection, Technical Report, Computer Science Dept., University of Nevada, Reno, 2001.

Liang, L., Basallo, E., and Looney, C., Image edge detection with fuzzy classifier, *Proc. ISCA 14th Int. Conf. CAINE-01,* Las Vegas, 2001, pp. 279–284.

Looney, C. G., Interactive clustering and merging with a new fuzzy expected value, *Pattern Recognition Lett.*, 35, 187–197, 2002.

Looney, C. G., Radial basis functional link nets and fuzzy reasoning, *Neurocomputing*, 2001a.

Looney, C. G., A Fuzzy Classifier Network with Ellipsoidal Epanechnikovs, Technical Report, Computer Science Dept., University of Nevada, Reno, 2001b.

Looney, C. G., Nonlinear rule-based convolution for refocusing, *Real-Time Imaging*, 6, 29–37, 2000.

Looney, C., *Pattern Recognition Using Neural Networks*, Oxford University Press, New York, 1997.

MacQueen, J. B., Some methods for classification and analysis of multivariate observations, *Proc. 5th Berkeley Symp. Probability Statistics*, University of California Press, Berkeley, 1967, pp. 281–297.

Maturino-Lozoya, H., Munoz-Rodriguez, D., Jaimes-Romero, F., and Tawfik, H., Handoff algorithms based on fuzzy classifiers, *IEEE Trans. Vehicular Technol.*, 49(6), 2286–2294, 2000.

Pao, Y. H., Park, G. H., and Sobajic, D. J. Learning and generalization characteristics of the random vector functional link net, *Neurocomputing*, 6, 163–180, 1994.

Pena, J. M., Lozano, J. A., and Larranago, P., An empirical comparison of four initialization methods for the *k*-means algorithm, *Pattern Recognition Lett.,* 20, 1027–1040, 1999.

Russo, F., A new class of fuzzy operators for image processing, *IEEE Int. Conf. Neural Networks*, 1993, pp. 815–820.

Russo, F., A user-friendly research tool for image processing with fuzzy rules, *Proc. First IEEE Int. Conf. Fuzzy Sys.*, San Diego, 1992, pp. 561–568.

Russo, F. and Ramponi, G., Fuzzy operator for sharpening of noisy images, *IEEE Electron. Lett.*, 28, 1715–1717, 1992.

Schneider, M. and Craig, M., On the use of fuzzy sets in histogram equalization, *Fuzzy Sets Systems,* 45, 271–278, 1992.

Selim, S. Z. and Ismail, M. A., *k*-means type algorithms: a generalized convergence theorem and characterization of local optimality, *IEEE Trans. Pattern Analysis Machine Intelligence*, 6, 81–87, 1984.

Snarey, M., Terrett, N. K., Willet, P., and Wilton, D. J., Comparison of algorithms for dissimilarity-based compound selection, *J. Mol. Graphics Modeling*, 15, 3782–3785, 1997.

Specht, D. F., Probabilistic neural networks for classification, mapping or associative *memory, Proc. IEEE Int. Conf. Neural Networks*, 1, 525–532, 1988.

Specht, D. F., Probabilistic neural networks, *Neural Networks*, 1(3), 109–118, 1990.

Wedding, D. K., II and Cios, K. J., Certainty factors versus Parzen windows as reliability measures in RBF networks, *Neurocomputing*, 19, 151–165, 1998.

Xie, X. L. and Beni, G., A validity measure for fuzzy clustering, *IEEE Trans. Pattern Analysis Machine Intelligence*, 13(8), 841–847, 1991.

Yager, R. R. and D. P. Filev, Approximate clustering via the mountain method, *IEEE Trans. Sys.*, *Man Cybernetics,* 24, 1279–1284, 1991.

Further Information

For information on statistical pattern recognition, see Duda, R., Hart, P., and Stork, D., *Pattern Classification*, Second Edition, Wiley-Interscience, New York, 2001.

Bayesian (belief) networks can be applied to pattern recognition. For these networks, see Jensen, F., *Bayesian Networks and Decision Graphs*, Springer, New York, 2001.

For PR fundamentals (statistical, syntatic, graphical, neural network and other), see Looney, C. G., *Pattern Recognition Using Neural Networks,* Oxford University Press, New York, 1997; Chen, C. H., Pau, L. F., and Wang, P. S., Eds., *Handbook of Pattern Recognition & Computer Vision*, World Scientific, Singapore, 1999.

Theory and applications can be found in the following journals: *Pattern Recognition* (Elsevier), *Pattern Recognition Letters* (Elsevier), and *Transactions on Pattern Analysis and Machine Intelligence* (IEEE).