Special Software Development for Neural Network and Fuzzy Clustering Analysis in Geological Information Systems

by

Prof. Carl G. Looney and Han Yu
Computer Science Department
University of Nevada
Reno, NV 89557

Executive Summary.

These algorithms perform: i) a new type of fuzzy clustering to partition data sets into classes that either contain a mineral or do not; and ii) a new type of neural network training on mineral data sets so as to recognize the presence or absence of a mineral of interest. The software module was developed for use with the GIS ARCView software package, bit it can be run independently as well.

1. Introduction

The purpose of this project was to develop a software module for use with Arc-SDM. The module implements: i) a fuzzy clustering algorithm for unsupervised (self-organizing) classification of data; and ii) a neural network algorithm for training on known data so that it could then classify unknown data. Both algorithms are new and are not yet in use in the technical community. The software module performs data analysis in combination with Arc-SDM and with other software from other sources. This project, however, concerns only the development of the software to perform the fuzzy clustering on data and to train and apply radial basis functional link net neural networks to data, where the data was to be in the format required for Arc-SDM.

The programming was performed by Mrs. Xiangyin Yin and Mr. Han Yu, who worked as research assistants under the direction of Prof. Carl G. Looney in the Computer Science Department, University of Nevada, Reno, NV 89557.

This report includes the description of the two algorithms, respectively, for the fuzzy clustering and for the radial basis functional link net. It also includes the operation of the software.

2. The Fuzzy Clustering Algorithm

2.1 Classification and Clusters. The *classification* of a set of entities by a learning system based on their vectors of features (measured attribute values) is a powerful tool for acquiring knowledge from observations stored in data files. Given a set of *feature* vectors, a *self-organizing* process *clusters* them into classes with similar feature values, either from a *top-down* (*divisive*) or *bottom-up* (*agglomerative*) process. Classes can also be learned under a *supervised training* process where each input vector is labeled and the process parameters are adjusted until the process output for each input vector matches the target output (label, or codeword). Artificial neural networks learn with labeled data under supervised learning. In this section we develop an agglomerative self-organizing (unsupervised), or clustering, approach.

The major problems in clustering are: i) how do we find the number K of clusters for a given set of feature vectors $\{\mathbf{x}^{(q)}: q = 1,...,Q\}$ when K is unknown? ii) how do we assess the validity of a given clustering of a data set into K clusters? iii) how do we permit clusters to take their own natural shape rather than forcing them into the shapes of normed unit balls? iv) how do we assure that the clustering is independent of the order in which the feature vectors are input? and v) how do we assure the independence of the order in which clusters are merged? In what follows, $\{\mathbf{x}^{(q)}: q = 1,...,Q\}$ is a set of Q feature vectors that is to be partitioned (clustered) into K clusters $\{C_k: k=1,...,K\}$. For each k^{th} cluster C_k its *prototype* (or *center*) is designated by $\mathbf{c}^{(k)}$. The dimension of the feature vectors $\mathbf{x}^{(q)} = (x_1^{(q)},...,x_N^{(q)})$ and centers $\mathbf{c}^{(k)} = (c_1^{(k)},...,c_N^{(k)})$ is N. The feature vectors are standardized independently in each component so that the vectors belong to the N-cube $[0,1]^N$.

2.2 Clustering Validity. A ball of uniformly distributed vectors has no cluster structure. But if a set of vectors falls into multiple subsets that are compactly distributed about their centers and the centers are relatively far apart, then there is strong cluster structure. We say that such clusters are *compact* and *well-separated*. These are criteria for well clustered data sets when the clusters are contained in nonintersecting normed balls. Not all data sets have strong cluster structure.

A *clustering validity measure* [12, 19] provides a goodness-of-clustering value. Some general measures for fuzzy clusters are the *partition coefficient* (PC), the *classification entropy* (CE) and the *proportional exponent* (PE). The *uniform distribution functional* (UDF) is another measure that was proposed in 1982. These are not as independent of K as is the Xie-Beni [19] clustering validity that is a product of compactness and separation measures. The *compactness-to-separation ratio* ν is defined over K clusters by

$$\nu = \{(1/K)\Sigma_{(k=1,K)}\, \sigma_k^{\,2}\}/\{D_{min}\}^2; \quad \sigma_k^{\,2} = \Sigma_{(q=1,Q)}\, w_{qk} \left\| \mathbf{x}^{(q)} - \mathbf{c}^{(k)} \right\|^2, \quad k = 1,...,K \tag{1}$$

where D_{min} is the minimum distance between the prototypes (cluster centers) and $\sigma_k^{\,2}$ is the (fuzzy) weighted variance of the k^{th} cluster over all Q vectors (even those in other clusters, which have very little influence). A larger D_{min} value means a lower reciprocal value. Any $\sigma_k^{\,2}$ is smaller for more compact clusters. Thus a lower value of ν means more compactness and greater separation.

We modify the Xie-Beni fuzzy validity measure by summing over only the members of each cluster rather than over all Q exemplars for each cluster. We also take the reciprocal $\kappa = 1/\nu$ so that a larger value of κ indicates a better clustering and we call κ the *modified Xie-Beni clustering validity measure.*

2.3 The New Fuzzy Clustering Algorithm. The *weighted fuzzy expected value* (WFEV) of a set of real values $x_1,..,x_P$ was defined by Schneider and Craig [15] to be a prototypical value that is more representative of the set than is either the average or the median. It uses the two-sided decaying exponential but we use the bell shaped Gaussian function instead in Equation (3) below because it is a canonical fuzzy set membership function for the linguistic variable *CLOSE_TO_CENTER*. After computing the arithmetic mean $\mu^{(0)}$ of a set of real values $x_1,..,x_P$ as the initial center value, we employ Picard iterations to obtain the new value

$$\mu^{(r+1)} = \Sigma_{(p=1,P)}\, \alpha_p^{(r)} x_p, \qquad (\sigma^2)^{(r+1)} = \Sigma_{(p=1,P)}\, \alpha_p^{(r)} (x_p - \mu^{(r)})^2 \tag{2}$$

$$\alpha_p^{(r)} = \exp[-(x_p - \mu^{(r)})^2/(2\sigma^2)^{(r)}] \,/\, \{\Sigma_{(m=1,P)} \exp[-(x_m - \mu^{(r)})^2/(2\sigma^2)^{(r)}]\} \tag{3}$$

The value $\mu = \mu^{(\infty)}$ to which this process converges is our *modified weighted fuzzy expected value* (MWFEV). Usually 5 to 8 iterations are sufficient for 4 or more digits of accuracy. For vectors the MWFEV is computed componentwise. The σ^2 in Equations (2, 3) is the *weighted fuzzy variance* (WFV). It changes over the iterations also and an initial value (for the *spread parameter* σ) can be set at 1/4 of the average distance between cluster centers. An open question currently is that of guaranteed convergence of the MWFEV and WFV. Our preliminary investigation shows empirically that the WFV can oscillate with increasing iterations, but that a subsequence converges (this is true of the fuzzy c-means weights). Therefore we substitute a fixed nonfuzzy mean-square error (variance) for the WFV in our algorithm.

The MWFEV method weights outliers less but weights the central and more densely distributed vectors more heavily. It provides a more typical vector to represent a cluster. Figure 1 at the end of Section 2 shows an example of the MWFEV versus the mean and the median for a simple 2-dimensional example. The 5 vectors are (1,2), (2,2), (1,3), (2,3) and (5,1). The outlier (5,1) influences the mean (2.2,2.2) and the median (2,2) too strongly. The MWFEV vector (1.503,2.282), however, is influenced only slightly by the outlier in the y-direction and even less in the x-direction.

We first standardize the Q feature vectors of a sample by affinely (linearly plus translation) mapping all Q values of the same component into the interval [0,1] (we do this independently for each n^{th} component over n = 1,...,N). The standardized vectors are thus in the unit N-cube $[0,1]^N$. We circumvent the problem that the order of input vectors will affect the clustering by using a large number K of uniformly distributed initial prototypes to spread out the initial cluster centers.

To cluster the sample $\{\mathbf{x}^{(q)}: q=1,...,Q\}$ we employ a relatively large initial number K of uniformly randomly drawn prototypes to start with a large number of initial cluster centers. Then we thin out the prototypes that are not useful and do our special fuzzy clustering, followed by fuzzy merging. The initial large number K and the thinning to a much smaller number ensures that we find reasonable centers for all clusters. The large initial K is empirically derived as shown in Table 1 by $K = \max\{6N + 12\log_2 Q, Q\}$. For example, for N = 32 and Q = 256 the computed K would be 288. In the main loop we compute the fuzzy weights and the MWFEV (componentwise) for each k^{th} cluster to obtain the prototype $\mathbf{c}^{(k)}$. We also compute the mean-squared error σ_k^2 for each k^{th} cluster. Then we again assign all of the feature vectors to clusters based on the minimum distance assignment. This process is repeated until the fuzzy centers do not change. The fuzzy clustering algorithm follows.

Fuzzy Clustering Algorithm

```
Step 1:  I = 0; input I_0;                    /*Set current and max. iteration
                                               numbers. */
         input N and Q;                       /*N =  no. components;  Q = no. feature vectors.
                                               */
         compute K;                           /* K = max{ 6N  + 12 log_2 Q, Q}. */
       for q = 1 to Q do  input x^(q);        /*Read in Q feature vectors in file to be
                                               clustered.*/
         for k = 1 to K do                    /* Generate uniform random prototypes in [0,1]^N
                                               .*/
             c^(k) = random-vector();
         for k=1 to K do  count[k] = 0;       /*Initialize number of vectors in each cluster*/
         τ = 1.0/(K)^(1/N);                   /*Threshold for prototypes being too close. */
Step 2:  repeat
        dmin = 9999.9;                        /*Parameter for smallest distance between
                                               centers. */
          for k=1 to K-1 do                   /*Check all distances between prototypes */
            for kk=k+1 to K do                /*to obtain minimum such distance. */
              if (‖c^(k) - c^(kk)‖ < dmin) then
                    dmin = ‖c^(k) - c^(kk)‖;
                    kmin = kk;                /*Index for possible deletion. */
          if (dmin < τ) then                  /*If deletion criterion is met then */
            for k= kmin to K-1 do
                                              /*remove prototype too close to another
                                               one */
              c^(k) = c^(k+1);                /*by moving all higher prototypes down
                                               one place. */
            K = K-1;                          /*Decrement K upon each removal. */
            again = true;                     /*Do process again if prototype is too close. */
          else again = false;                 /*Do not repeat if smallest distance is too large
                                               */
         until (again = false);               /*in which case "again" is a false boolean
                                               variable. */
```

```
Step 3:  for i = 1 to 20 do                    /*Do 20  k-means iterations. */
            for q = 1 to Q do                  /*Assign each x^(q) via  nearest prototype  c^(k). */
               k* = nearest(q);                /*c^(k*) is nearest prototype to x^(q). */
               clust[q] = k*;                  /*Assign vector x^(q) to cluster k*. */
               count[k*] = count[k*]+1;        /*Update count of cluster k*. */
            for k=1 to K do
               c^(k) = (0,...,0);              /*Zero out all prototypes (centers) for averaging.
                                                  */
               for q = 1 to Q do               /*For each feature vector, check if it belongs */
                  if (clust[q] = k)            /*to cluster k, and if so, then add it to the sum*/
                     c^(k) = c^(k) + x^(q)     /*averaging to get new prototype
                                                  (componentwise). */
                  c^(k) = c^(k)/count[k];
Step 4:  eliminate(p);                         /*Eliminate clusters of count ≤ p (user given p).
                                                  */
Step 5:  for k = 1 to K do                     /*Begin fuzzy clustering. */
            fuzzyweights();                    /*Use Eqn. (16), compute fuzzy wts. in k^th
                                                  cluster. */
            σ_k^2 = variance();                /*Get variance (mean-square error) of each
                                                  cluster. */
            c^(k) = fuzaverage();              /*Fuzzy prototypes (Eqn. 15a. */
         for k=1 to K do  count[k] = 0;        /*Zero out counts for reassignment.*/
Step 6:   for q = 1 to Q do                    /*Assign each x^(q) to a C_k via minimal distance.
                                                  */
            k* = nearest(q);                   /*Assign x^(q) to nearest c^(k*). */
            clust[q] = k*;                     /*Record this with index clust[q] = k*. */
            count[k*] = count[k*]+1;           /*Increment count for assigned vector. */
         I = I + 1;                            /*Increment iteration number. */
Step 7:  if (I > I_0) then                              /*Check for stopping criterion. */
            compute κ;                         /*Compute modified Xie-Beni validity measure.
                                                  */
            merge();                           /*Call function to merge, eliminate empty
                                                  results. */
            stop;                              /*Stop if stop_criterion = TRUE*/
         else go to Step 5;                    /*or if number of iterations is exceeded, else
                                                  repeat loop*/
```

```
Elimination Algorithm (eliminate(p) in Step 4)   /*For each cluster k, eliminate it if its */
                                                        /*count ≤ p (empty clusters also). */
         for k = 1 to K-1 do
            if (count[k] ≤ p)          then      /*First, test for size < p. If so, then move*/
               for i = k to K-1 do               /*all count indices down by 1. */
                  count[i] = count[i+1];
```

```
                    for q = 1 to Q do          /*Move assignment indices down by 1 */
                        if (clust[q] = i+1) then /*for any vector in k^th cluster. */
                            clust[q] = i;
                    K = K - 1:                 /*Decrement no. clusters on elimination. */
                if (count[K] ≤ p) then K = K - 1;  /*Check to eliminate last cluster center. */
```

2.4 Fuzzy Merging of Clusters. We first find the R = K(K-1)/2 distances {d(r): r = 1,...,R} between unique pairs of prototypes. For each such distance, the indices k1(r) and k2(r) record the two cluster indices for which the distance d(r) was computed, ordered so that k1(r) < k2(r). Each r^{th} prototypical pair k1(r) and k2(r) is merged only if the following condition is met: d(r) < βD, where D is the WFEV of all of the distances {d(r): r = 1,...,R} and β satisfies 0.0 < β < 1.0, where β = 0.5 is a good empirically derived first trial value). There are optional criteria as well. One is: the ball of radius $(\sigma_{k1(r)} + \sigma_{k2(r)})/2$ centered on the midpoint vector $\mathbf{y} = ½[\mathbf{c}^{(k1(r))} + \mathbf{c}^{(k2(r))}]$ between the cluster centers contains at least 20% of the vectors in each cluster (the default percentages can be changed).

Merging Clusters (merge())

```
    Step 1:  r = 1;                          /*For every one of the K(K-1)/2 unique */
             for k = 1 to K-1do               /*pairs of prototypes, get the distance*/
                 for kk = k+1 to K do         /*between them and get the indices for*/
                     d[r] = ‖c^(k) - c^(kk)‖;  /*the prototypical pairs (k1() and k2(). */
                     k1[r] = k; k2[r] = kk;
                     r ← r + 1;              /*Increment r (r^th pair), 1 ≤ r ≤ R=K(K-1)/2. */
             for k=1 to K do Merge[k] = 0;    /*Zero out merge indices. */
    Step 2:  D =  MWFEV({d[r]}_r);           /* Find MWFEV of the d(r) values. */
    Step 3:  r* = 1;
             for r = 2 to K(K-1)/2 do         /*Over all inter-prototype distances */
                 if d[r] < d[r*] then  r* = r;  /*find a pair with least distance for merge test.
                                                */
[   Step 4:  K1 = 0; K2 = 0;                 /*Optional: Zero out counters 1 and 2. */
             y = ½[c^(k1[r*]) + c^(k2[r*])];  /*Compute midway vector between prototypes.
                                                */
             σ = σ_k1(r*) + σ_k2(r*) ;       /*Compute σ for ball centered at y. */
             for q=1 to Q do                 /*Find feature vectors in overlapping ball. */
                 if (‖x^(q) - y‖ < σ) then   /*x^(q) must be close to y and also be in either */
                     if (clust[q] = k1[r*]) then K1 = K1+1; /*cluster k1(r*) or cluster k2(r*).
*/
                     if (clust[q] = k2[r*]) then K2 = K2+1;
             P1 = K1/count[k1[r*]];          /*Compute proportion of C_k1(r) points in ball, */
             P2 = K2/count[k2[r*];  ]        /*proportion of C_k2(r)  points in ball (optional).*/
    Step 5:  if  (d[r*] < βD)                /*Test merging criteria (try β = 0.5 first). */
```

```
[ AND  ((P1 ≥0.2 AND P2 ≥0.2) ]   /*Optional criteria for merging. */
        for q = 1 to Q do          /*Check each q^th feature vector: is it in C_k2(r*)? */
            if (clust[q] = k2[r*] then   /*If so, then reassign it to C_k1(r*) (Class k1(r*)). */
                clust[q] = k1[r*];       /*k2(r) > k1(r) is always true for same r. */
                count[k1[r*]] = count[k1[r*]]+1; /*Increment count of Class k1(r)*/
                count[k2[r*]] = count[k2[r*]] - 1; /*Decrement count of Class k2(r)*/
            Merge[k2[r*]] = [k1[r*]];    /*Record merger. */
            z^(k2[r*]) = c^(k2[r*]);     /*Save old prototype. */
            stop_criterion = false;
        else
            stop_criterion = true;
Step 6: eliminate(0);              /*Eliminate empty clusters (p = 0). */
        if (stop_criterion) stop;  /*Test for stopping( no previous merging). */
        else goto Step 1;          /*If no stopping then repeat merge test. */
```

This algorithm merges successfully with or without the optional criteria in Steps 4 and 5. When cluster k2(r) is merged with cluster k1(r) we record that as *Merge*(k2(r)) = k1(r). It is always true that k1[r] < k2[r].

2.5 Computer Test Results. To show the affects of ordering on the standard (Forgy [3] and MacQueen [10]) k-means algorithm we used the data file *testfz1a.dta* shown in Figure 2 (at the end of Section 2). The numbers 1 through 15 are the vector indices that give their order of presentation to the algorithm. The file *testfz1b.dta* was formed by exchanging feature vector number 1 with feature vector number 8 in the ordering. With K = 5 the different results are shown in Figures 3 and 4. Other changes in the ordering caused yet other clusterings. The shown clusterings are not intuitive. Our merging algorithm applied to the clusters in both Figures 3 and 4 with $\beta = 0.5$ yielded the identical results shown in Figure 5, which agrees more with our intuition of the classes. It is known that the k-means algorithm convergs to a local minimum [16], but it may not be a global minimum.

Figure 6 shows the respective results of our new fuzzy clustering and fuzzy merging algorithms on the file *testfz1a.dta*. The results were the same for *testfz1b.dta*. The modified Xie-Beni validity values were also identical for the two files. This, and other tests, showed that the strategy of drawing a large number of uniformly distributed prototypes and then thinning them out to achieve a smaller number of more sparsely uniformly distributed prototypes prevented the order of inputs from affecting the final clustering in these cases. The initial K was 59, which was reduced to 8 by eliminating empty clusters and then to 5 with p = 1. K was reduced to 3 by merging with $\beta = 0.5$. Figure 7 shows the K = 5 results.

The second data set is Anderson's [1] well-known set of 150 feature vectors of that are known to be noisy and nonseparable. The data is labeled as K = 3 classes that represent 3 subspecies (*Sestosa*, *Versicolor* and *Virginica*) of the iris species. The given sample contains 50 labeled feature vectors from each class for a total of Q = 150. The feature vectors have N = 4 features: i) sepal length; ii) sepal width; iii) petal length; and iv) petal width. Table 1 presents the results of our complete fuzzy clustering and fuzzy merging algorithm. Our modified Xie-Beni clustering validity measure shows that K = 2 classes are best, which coincides with the PC and CE validity values, but the PE gave the best value for K = 3 rather than K = 2 [6]. This yields 3 votes out of 4 that K = 2 is the best number of clusters for the iris data set. We have reduced the dimensions from 4 to 2 by taking ratios of petal width to length and sepal width to length, which reduces the noise due to size of the specimen and the plotted results show clearly that there are two classes. We note that in each clustering of Table 1 the *Sestosa* class contained the correct number of 50 feature vectors (underlined) and the same weighted fuzzy variance (underlined). The second class therefore has two subclasses that are not well separated.

Initially, our value for K was 150, but after deletions (due to close prototypes) it was reduced to 57. K was further reduced to 16, 9 and then 7 by elimination with p = 2, 6 and 10, respectively. After 40 fuzzy clustering iterations, the fuzzy merging with $\beta = 0.5$ reduced K to 4 as shown in Table 1. This was followed by further fuzzy merging with $\beta = 0.66$ to yield K = 3 clusters, and then with $\beta = 0.8$ to yield K = 2. The *modified Xie-Beni validities* are also shown in Table 1.

Table 1. Fuzzy Clustering/Merging the Standardized Iris Data.

Data File	No. Clusters K	Validity	Cluster Sizes	σ_k
iris150.dta	4	0.5766	5*0*, 40, 29, 31	0.192, 0.174, 0.182, 0.227
	3	1.1520	50, 51, 49	0.192, 0.248, 0.215
	2	***5.8216*** ←	50, 150	0.192, 0.323

Our third data set is taken from Wolberg and Mangasarian [17, 18] at the University of Wisconsin Medical School (also see [11]). We randomly selected 200 of the more than 500 feature vectors of that contained 30 features. As usual, we standardized each feature separately to be in [0,1]. The vectors are labeled for two classes that we think are *benign* and *malignant*. One label is attached to 121 vectors while the other is attached to 79 vectors. Our radial basis functional link net (neural network) "learned" the two groups of 121 and 79 vectors "correctly". However, the vectors do not naturally fall precisely into these groups because of noise and/or mislabeling (a danger in supervised learning with assigned labels). Table 2 shows that K = 2 (with sizes of 128 and 72) has the best clustering.

Table 2. Results on the Wolberg-Mangasarian Wisconsin Breast Cancer Data.

Data File	No. Clusters K	Validity	Cluster Sizes
wbcd200.dta	5	0.1050	30, 53, 69, 22, 26
	4	0.2575	35, 117, 22, 26
	3	0.4114	54, 109, 37
	2	***1.1442*** ←	128, 72

The original K was 272, which was not reduced by deletion of close clusters with the computed τ (N = 30 features provided a large cube). Eliminations with p = 1, 7, and 11 and fuzzy clustering and merging with β = 0.5, 0.66, 0.8 and 0.88 reduced K 5, 4, 3, and 2, respectively. We conclude from the validity values that the data contains extraneous noise and does not separate into compact clusters.

The fourth data set was geological data provided by Dr. Gary Raines of the US Geological Survey. It is labeled for K = 2 classes. Each data vector has 4 feature values and a fifth component that is the class label. There were Q = 70 feature vectors in the noisy sample where the labels were made by humans who assigned 35 vectors to one class and 35 to the other. Our neural network "learned" the labeled classes "correctly," which shows that neural networks learn what they are instructed to learn, whether or not it is correct. The clustering showed that the data did not fall into equal sized classes.

For fuzzy clustering, the large initial number K of class prototypes was 98, but this was reduced to 33 by deletion of close prototypes and then respectively reduced to 13 and 8 by elimination with p =1 and p = 4. After 40 fuzzy clustering iterations, the fuzzy mergings with β = 0.5, 0.66, 0.8 and 0.88 brought it down to the respective K values of 7, 5, 3 and 2. Other paths through the deleting, eliminating and merging also yielded the same results as are shown in Table 3..

Table 3. Results on the Geological Labeled Data.

Data File	No. Clusters K	Validity	Cluster Sizes
geo70.dta	5	0.6267	14, 26, 7, 16, 7
	3	1.0112	17, 13, 40
	2	**2.1545** ←	51, 19

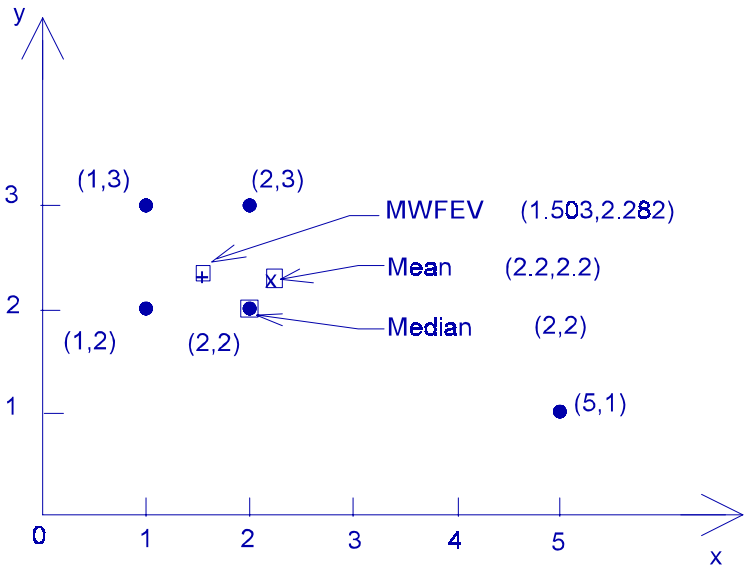


Figure 1. MWFEV vs. mean and median.

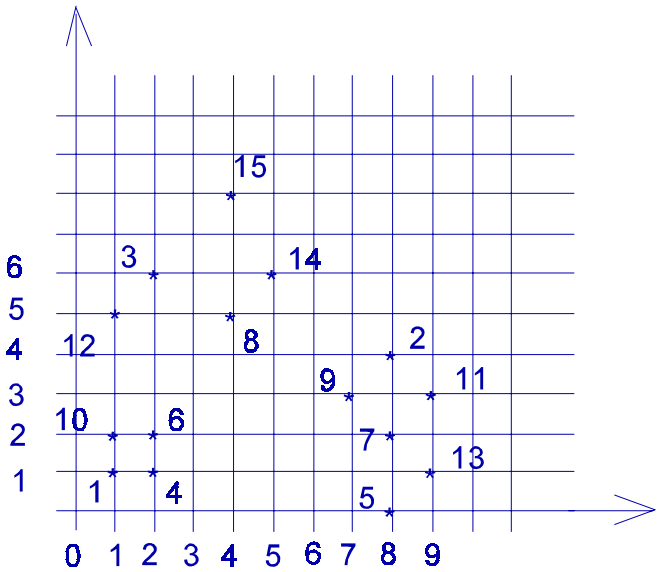


Figure 2. Original sample.

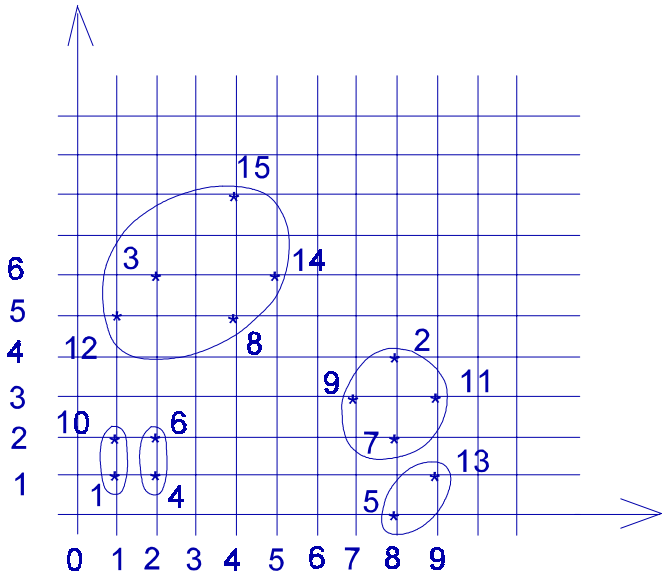


Figure 3. K-means, original order, K = 5.

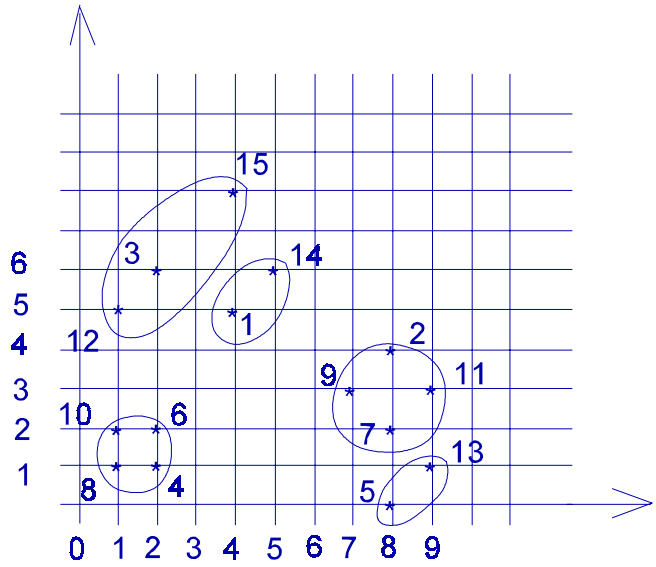


Figure 4. K-means, 1 and 8 exchanged, K = 5.

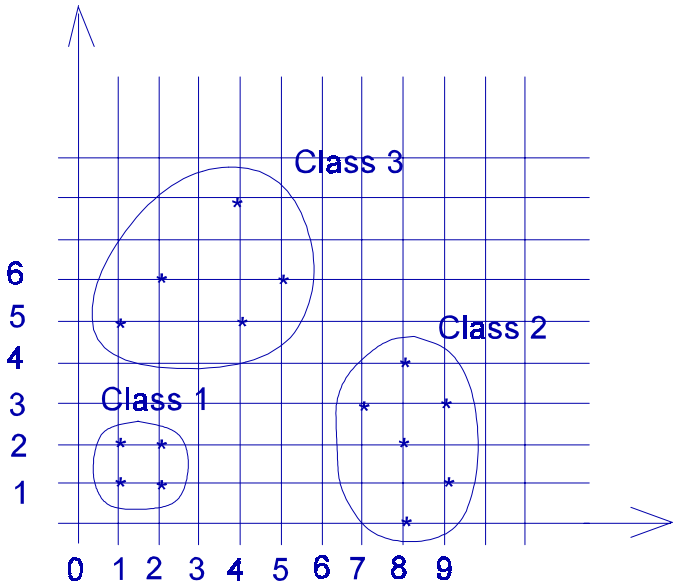


Figure 5. Fuzzy merging of k-means clusters.

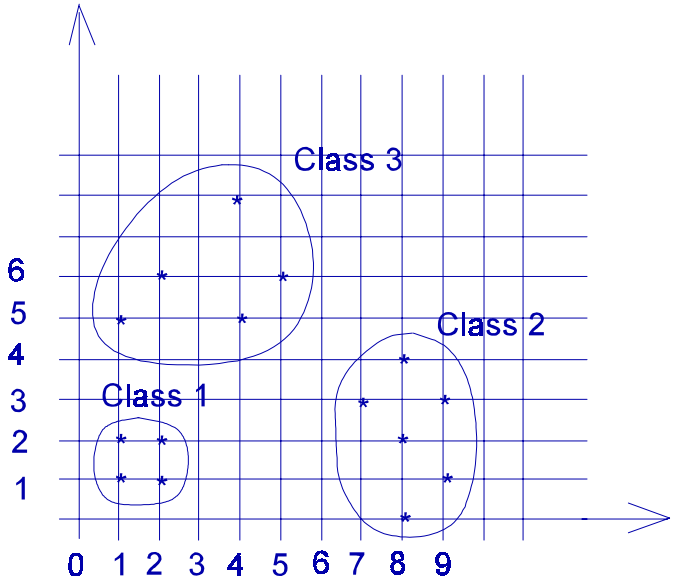


Figure 6. Fuzzy clustering/merging.

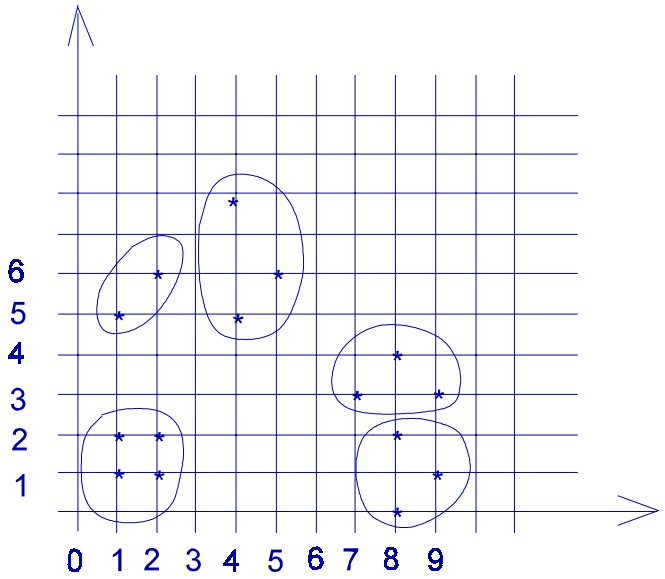


Figure 7. New method, thinned to K=5.

3. The Radial Basis Functional Net Algorithm

3.1 Radial Basis Functions. *Radial basis functions* (RBFs) originated in 1964 as potential functions [7]. The architecture and training algorithms for RBFNNs are simple and they train more quickly than do multiple layered perceptron (MLP) networks. Unlike MLPs, they allow for explanation when interpreted as fuzzy rule-based systems. We use RBFs with the *random vector functional link nets* (RVFLNs) of Pao et al. [13] to obtain a more general and powerful *radial basis functional link net* (RBFLN).

Figure 8 presents a *radial basis function* (RBF) on the vectors **x** in the plane. An RBF with center vector at **v** is generally defined on N-dimensional feature vectors **x** by

$$y = f(\mathbf{x};\mathbf{v},\sigma) = \exp[-\|\mathbf{x} - \mathbf{v}\|^2/(2\sigma^2)] \qquad (4)$$

with *spread parameter* σ. The name *radial* indicates that all points **x** equidistant from **v** yield the same value y. A number M of these radial basis functions can be centered on M center vectors so that the circular balls of radius σ_m cover a bounded region of interest in the feature space. We consider **v** as the prototype, or center, of a cluster of radius σ-ball.

Figure 9 shows a slice through the center of an RBF and its *receptive field*, also called the *response region*. Figure 10 presents an RBFNN that implements the composite mapping **x** → **y** → **z**. Each node in the *hidden* (center) layer represents an RBF. An input feature vector **x** *activates*

(*fires*) a unique vector **y**. The weights $\{u_{mj}\}$ at the nodes in the output layer are trained (adjusted) to force the outputs **z** to approximate their targets **t** very closely (the targets are the labels that humans supply). *Generalized Gaussian basis functions* with inverse covariance matrices C^{-1} may be used in hidden nodes, especially where C is diagonal so that the diagonal of C^{-1} contains the reciprocals of the diagonal elements of C. Such elliptical Gaussian RBFs are radial with respect to the *Mahalanobis* distance (e.g., see [7]) and satisfy

$$y = \exp[-(\mathbf{x}-\mathbf{v})^t C^{-1}(\mathbf{x}-\mathbf{v})] \tag{5}$$

3.2 Radial Basis Functional Link Nets. The RBFNN of Figure 10 has an *input layer* of N nodes, a *hidden layer* of M *neurodes* (neural nodes rather than biological neurons) and an *output layer* of J neurodes. Feature vectors **x** are fed to the input layer. The respective outputs from the m^{th} neurode of the hidden layer and from the j^{th} node of the output layer for the q^{th} input exemplar vector $\mathbf{x}^{(q)}$ are

$$y_m^{(q)} = \exp[-\|\mathbf{x}^{(q)} - \mathbf{v}^{(m)}\|^2/(2\sigma_m^2)], \qquad z_j^{(q)} = (1/M)[\Sigma_{(m=1,M)} u_{mj} y_m^{(q)} + b_j] \tag{6a,b}$$

where m = 1,...,M and j = 1,...,J. The weights u_{mj} are *gains* on the lines from the hidden layer to the output layer. The usual *bias* b_j is included at each output node as shown in Figure 10 to model translations.

Figure 11 presents three types of feedforward neural networks: a) a multiple layered perceptron (MLP) with sigmoid functions in the hidden and output layers; b) an RBFNN with RBFs in the hidden layer and sums in the output layer; and c) our *radial basis functional link net* (RBFLN) with extra lines from the input nodes that connect directly to the output neurodes with another set of weights $\{w_{nj}\}$ on these lines (w_{nj} weights x_n at the j^{th} output node). This is an RBFNN when the extra weights are set to zero.

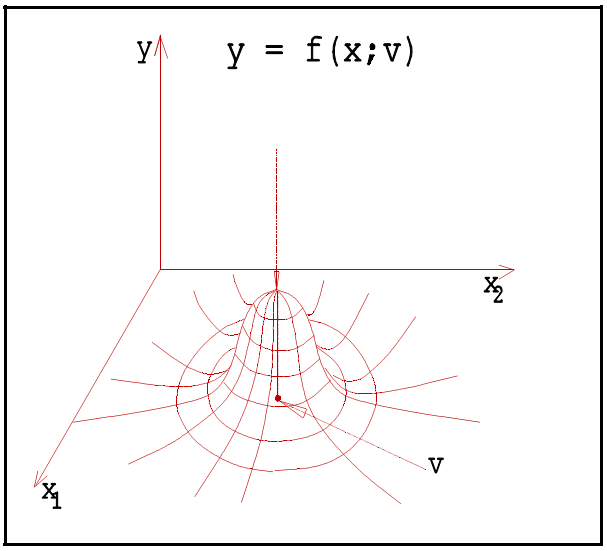


Fig. 8. An RBF defined cluster center.

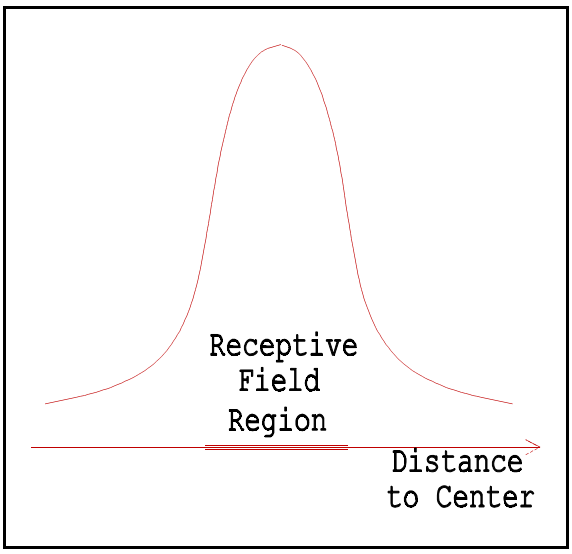


Fig. 9. A radial basis function slice.

An RBFNN represents a nonlinear model while our RBFLN includes that nonlinear model as well as an additive linear (affine) model due to the direct lines from the input to the output nodes. Such parts of a transformation being modeled do not need to be approximated by the nonlinear RBFNN. Thus the RBFLN is a more complete model of a general nonlinear mapping than is the RBFNN. Multiple layered perceptrons (MLPs) [9] and RBFNNs, and thus RBFLNs (which include RBFNNs) are universal approximators [4, 5, 14] and can approximate any mappingof N-dimensional space into J-dimensional space as closely as desired provided that a sufficiently large number of neurodes is used.

The training is done with a sample of Q input *exemplar* feature vectors and a set of Q associated output *target* vectors, designated respectively by

$$\{\mathbf{x}^{(q)}: q = 1,...,Q\}, \qquad \{\mathbf{t}^{(q)}: q = 1,...,Q\}$$

The training of an RBFLN consists of the following two stages: i) *initialization* of the centers, spread parameters and weights; and ii) *weight and parameter adjustment* to minimize the output *total sum-squared error* (TSSE) E defined as the sum of the *partial sum-squared errors* (PSSEs) in

$$E = \Sigma_{(q=1,Q)}\{E^{(q)}\} = \Sigma_{(q=1,Q)}\{\Sigma_{(j=1,J)}(t_j^{(q)} - z_j^{(q)})^2\} \tag{7}$$

The RBFLN output components differ from Equation (6b) for RBFNNs and are given by

$$z_j^{(q)} = [1/(M+N)]\ \{\Sigma_{(m=1,M)}u_{mj}y_m^{(q)} + \Sigma_{(n=1,N)}w_{nj}x_n^{(q)}\} \tag{8}$$

Training on the weights is extremely quick via steepest descent iteration per

$$u_{mj} \leftarrow u_{mj} - \eta_1(\partial E/\partial u_{mj}) = u_{mj} + (\eta_1/(M+N))\Sigma_{(q=1,Q)}(t_j^{(q)} - z_j^{(q)})y_m^{(q)} \quad (9)$$

$$w_{nj} \leftarrow w_{nj} - \eta_2(\partial E/\partial w_{nj}) = w_{nj} + (\eta_2/(M+N))\Sigma_{(q=1,Q)}(t_j^{(q)} - z_j^{(q)})x_n^{(q)} \quad (10)$$

Each center and spread parameter can also be updated with steepest descent via

$$v_n^{(m)} \leftarrow v_n^{(m)} - \eta_3(\partial E/\partial v_n^{(m)}) =$$

$$v_n^{(m)} + [\eta_3/\sigma_m^2]\Sigma_{(q=1,Q)}\{\Sigma_{(j=1,J)}(t_j^{(q)} - z_j^{(q)})u_{mj}\}y_m^{(q)}(x_n^{(q)} - v_n^{(m)}) \quad (11)$$

$$\sigma_m^2 \leftarrow \sigma_m^2 - \eta_4(\partial E/\partial(\sigma_m^2)) =$$

$$\sigma_m^2 + (\eta_4/\sigma_m^4)\Sigma_{(q=1,Q)}\{\Sigma_{(j=1,J)}(t_j^{(q)} - z_j^{(q)})u_{mj}\}[y_m^{(q)}\|\mathbf{x}^{(q)} - \mathbf{v}^{(m)}\|^2] \quad (12)$$

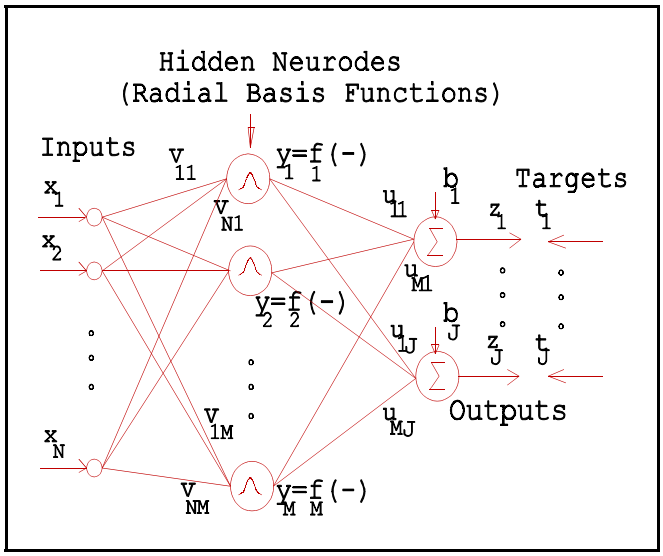


Fig. 10. An RBFNN.

The *learning rates* η_i absorb factors of ½ in certain of Equations (9-12). Instead of training over epochs with a single q^{th} input feature vector at a time as does backpropagation, we input all Q feature vectors and use steepest descent on the TSSE error E rather than on each PSSE in Equation (7). We call this technique *fullpropagation*. [8, 9].Our adjustment of each weight is influenced by all Q input feature training vectors, which is more stable and substantially more efficient than epochal training.

3.3 The Full Training Algorithm for RBFLNs. We present here the *full training* (FT) algorithm for RBFLNs where the centers and spread parameters are adjusted as are the weights. The *quick training* (QT) algorithm is included in this upon omitting the adjustments of the spread parameters and the RBF center vectors. FT and QT for RBFNNs are included when the lines from the input nodes to the output nodes have zero weights (all $w_{nj} = 0$). QT is fast with a single global minimum [2].

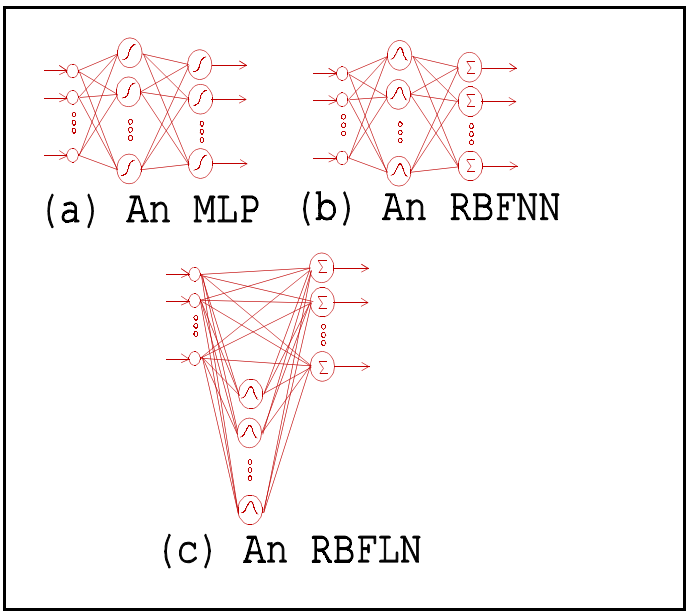


Fig. 11. Three feedforward architectures.

Full Training Initialization

Step 1: Given randomly ordered exemplar training vectors $\{\mathbf{x}^{(r)}: r = 1,...,R\}$ of dimension N, select Q < R vectors for training and save the R - Q remaining ones for testing (validation) of the training.

Step 2: Choose a large M so that there are M small balls of radius σ that essentially cover the feature space. Put $\mathbf{v}^{(m)} = \mathbf{x}^{(m)}$ if $M \leq Q$, else put $\mathbf{v}^{(q)} = \mathbf{x}^{(q)}$, q = 1,...,Q and draw the remaining M - Q centers at random in the feature space.

Step 3: Compute initial $\sigma = (1/2)[1/M]^{1/N}$ and put $\sigma_m = \sigma$ for m = 1,...,M.

Step 4: Select all weights $\{u_{mj}\}$ and $\{w_{nj}\}$ randomly between -0.5 and 0.5 for j = 1,...,J; m = 1,...,M; and
n = 1,...,N. Specify I iterations (e.g., I = 100).

Full Training Iteration

Step 5: Compute $y_m^{(q)}$ from Equation (6a) for each m = 1,...,M and q = 1,...,Q.

Step 6: Compute $z_j^{(q)}$ from Equation (8) for each j = 1,...,J and q = 1,...,Q.

Step 7: Update weights in Equation (9) and adjust the learning rate η_1 up on success, else down.

Step 8: Update weights in Equation (10) and adjust the learning rate η_2 up on success, else down.

Step 9: Update centers in Equation (11) and adjust the learning rate η_3 up on success, else down.

Step 10: Update spread parameter in Equation (12) and adjust the learning rate η_4 up on success, else down.

Step 11: Every P^{th} iteration (e.g., P = 20), put the test vectors through the network and record the *validation* TSSE E_{val}. If E_{val} increases for the first time and all training and test vectors are

mapped into the correct targets, then stop.

Step 12: If I iterations are done, then stop, else go to Step 5 above.

Some refinements yield improved performance. Our *en route* technique [9] adapts the *learning rates* $\{\eta_k\}$ of Equations (9 - 12) during iteration via an auto-adjustment mechanism that speeds up convergence considerably. It multiplies a learning rate η by a number greater than 1, say, 1.04, whenever a step of weight adjustments is successful in reducing the TSSE E, or else it multiplies η_k by a number less than 1, say 0.94. We let the learning rates η_1 and η_2 grow very large, which yields a *greedy* algorithm (it takes chances to converge quickly, but may have to recover if it jumps too far). In contrast, backpropagation algorithms must keep the learning rate small, which causes slow convergence.

Training on all Q input feature vectors on each iteration (fullpropagation) also increases the learning efficiency by eliminating *thrashing* (learning and unlearning over a sequence of PSSEs in backpropagation). QT omits Steps 9 and 10 and loops from Step 12 back to Step 6 rather than Step 5 because the outputs of the RBFs do not need to be recomputed (there are no changes in their values because neither the centers nor the spread parameters are changed). This yields a dramatic increase in convergence speed.

3.4 Comparative Training Runs on Noisy Data. In all of our runs the learning rates η_1 and η_2 were adjusted every 10^{th} iteration, as was the single spread parameter σ. The centers were not adjusted. Thus we used a *modified* QT for both RBFLN and RBFNN runs.

Table 4 presents the first data set that contains geological feature vectors from which explorations can be made. There are R = 69 feature vectors of N = 4 dimensions and each has an assigned target vector of a single component (0 or 1 for absence or presence of the desired substance). There were 70 original vectors but we deleted one because it affected the training adversely (it appears that it was too noisy and also had an incorrect label).

We selected 10 feature vectors at random to be used only for testing, which left Q = 59 feature vectors to be used for training. We used both M = 120 and M = 96 hidden neurodes for the training runs on this data. Q feature vectors were used as RBF centers and M - Q centers were drawn randomly as additional centers. The RBFLN was trained on the 59 training vectors for a number P of iterations and the training total sum-squared error (TSSE) E was recorded. Then the 10 test vectors were put through the RBFLN and the *validation* TSSE E_{val} and classification results were retrieved. This two step process of obtaining E and E_{val} was repeated for various numbers of iterations. No training was done on the test vectors, which were used only to show at what point the generalized learning was completed and the specialized learning [7] was starting (to minimize E_{val} approximately).

Table 4. Some Geological Data

x				t		x				t
0.6	0.6	0.2	0.2	1*		0.5	0.0	0.1	0.6	1
0.5	0.2	0.2	0.3	1		0.9	0.1	0.3	0.3	1
0.7	0.2	0.3	0.3	1		0.7	0.2	0.3	0.3	1
0.8	0.5	0.5	0.3	1*		0.8	0.5	0.5	0.3	1
1.0	0.6	0.7	0.3	1		1.0	0.7	0.7	0.3	1
1.0	0.7	0.7	0.3	1		1.0	0.7	0.7	0.3	1
1.0	0.3	1.0	0.3	1		1.0	0.5	0.9	0.3	1*
1.0	0.7	1.0	0.3	1		1.0	0.3	1.0	0.3	1
1.0	0.1	1.0	0.3	1		0.5	0.0	0.4	0.8	1
0.4	0.6	0.6	0.6	1		0.4	0.2	1.0	0.3	1
0.4	0.5	0.8	0.3	1*	0.6	0.0	0.2	0.3	1	
0.4	0.0	0.4	0.8	1		0.8	0.2	0.1	0.6	1
0.8	0.1	0.1	0.6	1		0.4	0.7	0.1	0.5	1
0.5	0.2	0.1	0.3	1		0.4	1.0	0.4	0.3	1*
0.4	0.6	0.5	0.3	1		0.2	1.0	0.2	0.7	1
0.2	0.9	0.3	0.7	1		0.2	1.0	0.3	0.7	1
0.2	0.8	0.1	0.6	1		0.6	0.7	0.3	0.7	1
0.4	0.0	0.2	0.5	0*	0.2	0.0	0.1	0.2	0	
0.2	0.0	0.1	0.5	0		0.2	0.9	0.1	0.2	0
0.2	0.0	0.1	0.8	0		0.2	0.0	0.1	0.2	0
0.2	0.0	0.1	0.5	0		0.2	0.0	0.1	0.5	0*
0.1	0.0	0.3	0.8	0		0.2	0.0	0.1	0.2	0
0.2	0.5	0.1	0.2	0		0.2	0.7	0.1	0.2	0
0.1	0.0	0.1	0.2	0		0.1	0.3	0.1	0.5	0
0.2	0.0	0.1	0.2	0*	0.2	0.4	0.2	0.2	0	
0.2	0.0	0.2	0.2	0		0.2	0.0	0.1	0.8	0
0.9	0.0	0.1	0.8	0		0.2	0.0	0.1	0.5	0
0.1	0.0	0.1	0.5	0		0.1	0.6	0.1	0.2	0*
0.2	0.0	0.1	0.2	0		0.2	0.7	0.1	0.5	0
0.4	0.0	0.1	0.5	0		0.2	0.0	0.1	0.5	0
0.2	0.6	0.1	0.2	0		0.1	0.0	0.1	0.2	0
0.2	0.0	0.1	0.2	0*	0.7	0.0	0.2	0.8	0	
0.4	0.0	0.2	0.5	0		0.2	0.0	0.1	0.2	0
0.2	0.4	0.1	0.8	0		0.2	0.0	0.1	0.5	0
0.2	0.0	0.1	0.8	0						

* used only for testing, not training

Table 5. RBFLN Runs on the Geological Data

Iterations	Training Error	Test-Error	σ^2	η_1	η_2	Training Misses	Test Misses
M = 120							
150	1.2397	0.4407	0.0526	292.67	805.19	0	0
180	1.0988	0.4107	0.0423	398.67	821.43	0	0
200←	**0.9863**	**0.4023←**	**0.0335**	**423.96**	**755.95**	**0**	**0←**
210	0.9247	0.4025	0.0304	543.09	838.00	0	0
230	0.8342	0.4433	0.0265	499.80	771.20	0	0
M = 96							
130	1.2835	0.4869	0.0418	238.15	490.70	0	0
140←	**1.2196**	**0.4770←**	**0.0404**	**228.46**	**543.96**	**0**	**0←**
160	1.0713	0.5013	0.0587	130.92	202.02	0	0

Table 5 presents the RBFLN results. The training on 200 iterations for M = 120 hidden neurodes gave the lowest TSSE E_{val} on the test feature vectors. The best generalized learning for M = 96 took 140 iterations. The output values were considered to be incorrect (misses) if they fell strictly between 0.42 to 0.58 or yielded the wrong class.

The famous *iris* data set contains 150 each of 4-dimensional feature vectors from 3 classes. We assigned the 3 target codewords (-1,0), (0,1) or (1,-1) for the respective classes 1, 2 and 3. These targets differ by at least one unit in each of two components. These feature vectors are known to be noisy and very difficult to classify. We extracted 18 vectors for a test set and trained on the remaining ones. Table 6 shows the results for our RBFLN and Table 7 shows the results for an RBFNN, both on the *iris* data. Neither the RBFLN nor RBFNN could learn with M = 60 RBFs, but did better as M increased until they did very well at M = 160. The RBFLN had a smaller TSSE than did the RBFNN and learned with fewer iterations. Our fullpropagation MLP had a difficult time on this data and did not learn all vectors correctly with any of M = 8, 12, 16, 20, 32 and 50.

Table 6. RBFLN Results on the Iris Data with M = 160

No. Iterations	Error	Test Error	Training Misses	Test Misses
480	1.1933	0.3620	0	0
500←	**1.1764**	**0 3603←**	**0**	**0←**
520	1.1578	0.3617	0	0

Table 7. RBFNN Results on the Iris Data with M = 160

No. Iterations	Error	Test Error	Training Misses	Test Misses
730	7.1979	0.6379	0	0
750←	**7.1130**	**0.6345←**	**0**	**0←**
770	7.0530	0.6356	0	0

The next data set is a subset of 200 normalized vectors taken from the Wisconsin Breast Cancer Data (Mangasarian et al. [11] and Wolberg et al. [17, 18]. The data we used has 30 features in each feature vector and a single output component as target that codifies two classes: benign and malignant (we set these to 0.1 and 0.9). We used a training file of 178 of these 200 vectors and a test file of 22 vectors. The respective results for RBFLNs, RBFNNs and MLPs are presented in Tables 8, 9 and 10.

Table 8. RBFLN Results on the WBCD Data

M	No. Iterations	TSSE	Test Error	σ	Training Misses	Test Misses
220	200	0.7658	0.6631	0.0643	2	0
220	**220←**	**0.6383**	**0.6789←**	**0.0587**	**0***	**0*←**
220	240	0.5473	0.6927	0.0549	0	0*

Table 9. RBFNN Results on the WBCD Data

M	No. Iterations	TSSE	Test Error	σ	Training Misses	Test Misses
220	300	(could not learn)				
260	300	(could not learn)				
300	300	(could not learn)				

Table 10. MLP Results on the WBCD Data

M	No. Iterations	TSSE	Test Error	Training Misses	Test Misses
20	1400	(could not learn)			
50	850	1.3773	0.0440	0	0
50	900	1.3437	0.0407	0	0
50	**950←**	**1.2736**	**0.0265←**	**0**	**0←**
50	1000	1.1704	0.0450	0	0

4. DataXplore Software Description

The software developed for user friendly running of these two algorithms was done in Visual C++ and is called *DataXplore*. Mr. Han Yu programmed the interfacing so that the user can select either the fuzzy clustering algorithm or the radial basis functional link network by clicking on the appropriate button in the main window. After selecting one of these, the user can click to select a file from a menu of directories and files. The parameters are set for automatic operation, but are shown in text boxes and may be changed by the user. In each case, when the user is ready to run the selected algorithm on the selected data, one clicks on the "Run" button.

The *DataXplore* system includes two major software modules, which respectively implement: i) a fuzzy clustering algorithm for unsupervised (self-organizing) classification of data; and ii) a neural network algorithm (the RBFLN) for training on known data so that it can then classify unknown data. The GIS package ARCView used by the Geological Survey of Canada can call external programs and DLLs and so can call the *DataXplore* program to perform data analysis on data in the format specified by the GS of Canada.

***4.1. Starting and Stopping* DataXplore.** The user can start *DataXplore* by either of two methods. First, from Windows Explorer, double-click *DataXplore.exe*. Second, select *Run* from the Windows *Start* menu, then type in the full path to the "DataXplore.exe" file or use *Browse* to find the location of the *DataXplore.exe* file, click on it and then click *OK*. Then select "Open" from the popup menu. The *Main Work Interface* will appear to prompt the user as shown in Fig. 12 below.

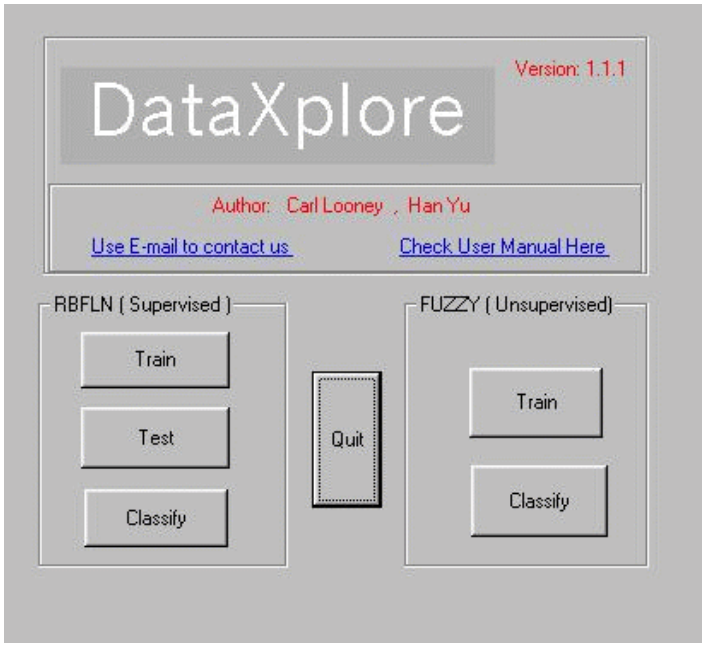
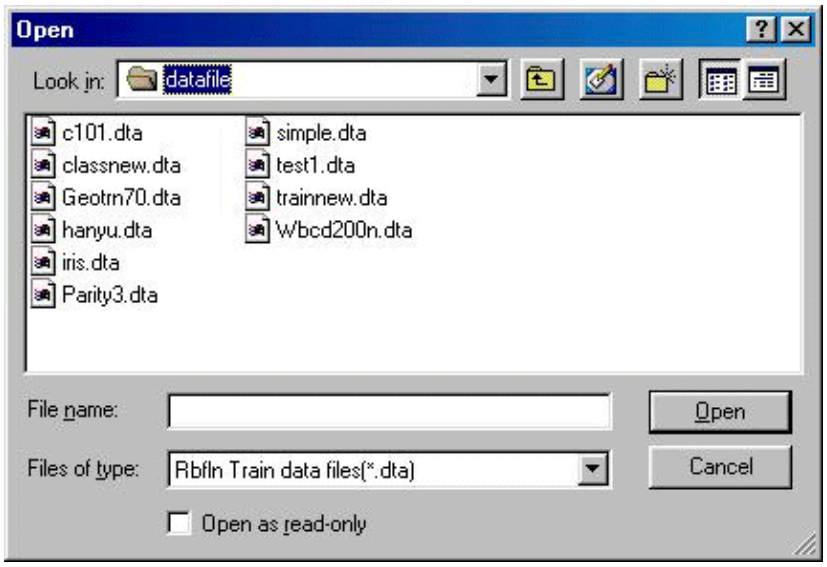


Fig. 12. The *Main Work Interface* menu.

Fig. 12 shows that *DataXplore* includes two major modules shown on the left and right of the *Main Work Interface*: i) *RBFLN* for radial basis functional link net supervised learning of data (on the left); and ii) the *FUZZY* unsupervised learning process (on the right). The details of each function are discussed in the following sections. The user can click on the *Quit* button in the center of the menu to exit the system.

4.2 The RBFLN (Supervised) Function. In Fig. 12, the *RBFLN* module includes three functions designated by the buttons with captions *Train*, *Test* and *Classify*. *Train* uses known data that includes both input vectors and target (labeled) vectors to train the RBFLN neural network so that it can perform automatic classification operation on unknown (unlabeled) vectors after training. *Test* is used after a partial or full training segment on the RBFLN network that yields a set of near optimal weights. It is used to classify a set of known (labeled) vectors to determine whether or not the training is good. These vectors have not been used in any training operations and are novel to the network and so provide a test. The *Classify* procedure uses the trained (and tested) weight parameters in the RBFLN in the on-line operational mode to process unknown data files of (unlabeled) vectors for the purpose of classifying them. After training and satisfactory testing, the RBFLN and its learned weights are called by the GIS to operate on other data sets of vectors with the feature values from the same population of objects as used in the training and testing to classify them into the appropriate classes that indicate the presence or absence of a desired mineral (or other uses).



Fig, 13. Selecting a data file for RBFLN training.

To train, the user clicks on the *Train* button of the RBFLN module (left side of the *Main Work Interface* menu of Fig. 12) to bring up the *OpenFile* dialog box of Fig. 13 below that prompts the user to select the data file to be used in the training stage (the default file name suffix

is *dta* for vector data sets). The user can browse and select the desired data file to use for training by clicking *Open* on the bottom right. A click on the *Cancel* button returns to the *Main Work Interface*.

A click on *Open* above brings up the RBFLN *Train Parameters* dialog box as shown in Fig. 14 to prompt the user for the content of the current initialization parameter set for the RBFLN training procedure and also to enable the user to make custom modifications. The upper part of this dialog box (Fig. 14) shows the file name in the *Training Data* text area and also shows four other parameters used to define the scale of the current RBFLN network. These are: *No. of Input Dimension*, *No. of Unique Conditions*, *No. of Hidden Layers* and *No. of Outputs*.

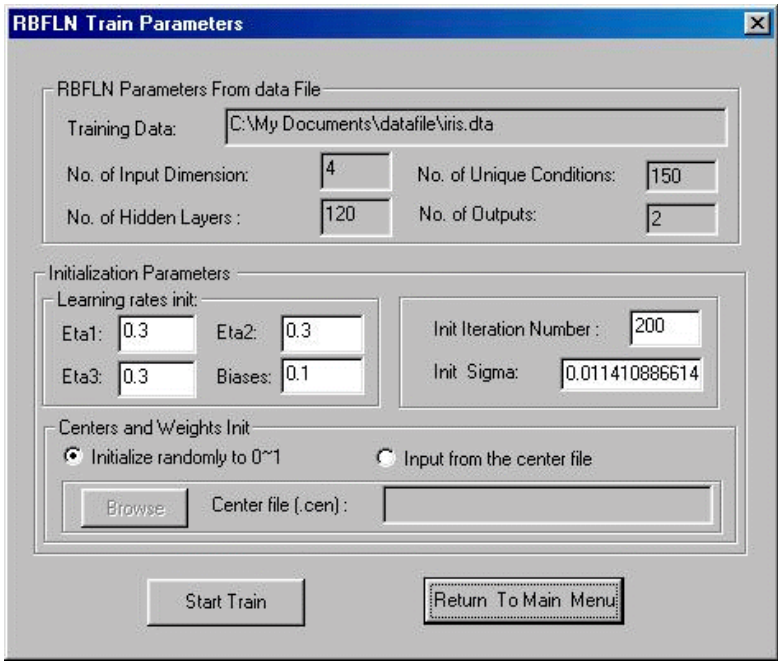


Fig. 14. The RBFLN *Train Parameters* dialog box.

The lower part of this dialog box permits the user to custom modify any initialization parameters, including the *learning rates* (eta1, eta2, eta3 and biases), *initial iteration number* and *initial sigma* values. The default values given are reasonable for essentially all cases and should usually be accepted. A user can also select which method in the *Centers and Weights Init.* Section toward the bottom will be used: *Initialize randomly to 0~1* or *Input from the center file* (the center file has a suffix of *.par* to denote that it is a file of parameters (weights and centers of the RBFs) learned from the previous training. The default configuration is the first (left) radio button selection and is used for initial training.

Using this dialog box, the user can define parameters once, then use them repeatedly for

later training, i.e., the user can make the training accumulative by the following operation: before the next segment of training, select *Input from the center file* and then locate the parameter *.par* file of the last training segment for the same training data file. During this procedure, the user can modify any other desired parameters to make the individual training stage more efficient.

In the *RBFLN Train Parameters* dialog box of Fig. 14, users can make parameter modifications based on previously completed training to start training again, or select "Return to Main Menu" to accept the current parameter file as optimal parameters for the RBFLN network so that it can be used for later "Test" and "Classify" stages.

After pressing *Start Train*, the *RBLFN Training* function starts. The *RBFLN Train Status* dialog box will pop up as shown in Fig. 15 below. Three areas are used to show the training status: i) *Status Description*, which gives textual information about which step of *RBFLN Training* is under processing; ii) *Current Iteration*; and iii) *Current SSE*, which is used to show the output's variance from the set of targets (the sum squared error of Equation (7)).

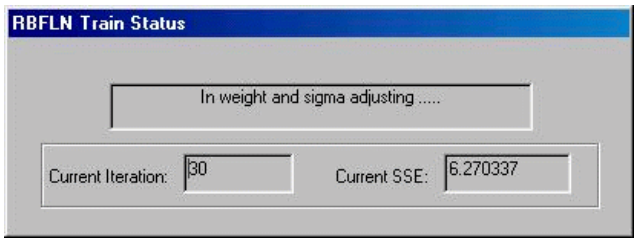


Fig. 15. The *RBFLN Train Status* dialog box.

After the training procedure finishes, the *RBFLN Train Result* dialog box of Fig. 16 comes up, which shows the RBFLN network general parameters, *SSE* for evaluating the results as the training proceeds, and *Vector No. / Target / Actual Output / Input* for each input feature vector. The user can print all mirrored content of the dialog box to a *txt*- format file. When the user decides that the result is optimal enough for use in classification of novel input feature vectors, one can save the parameter set to a parameter result file (with suffix *.par*) and return to RBFLN Train Parameters dialog for further training. The two buttons at the bottom of the dialog box allow these capabilities.

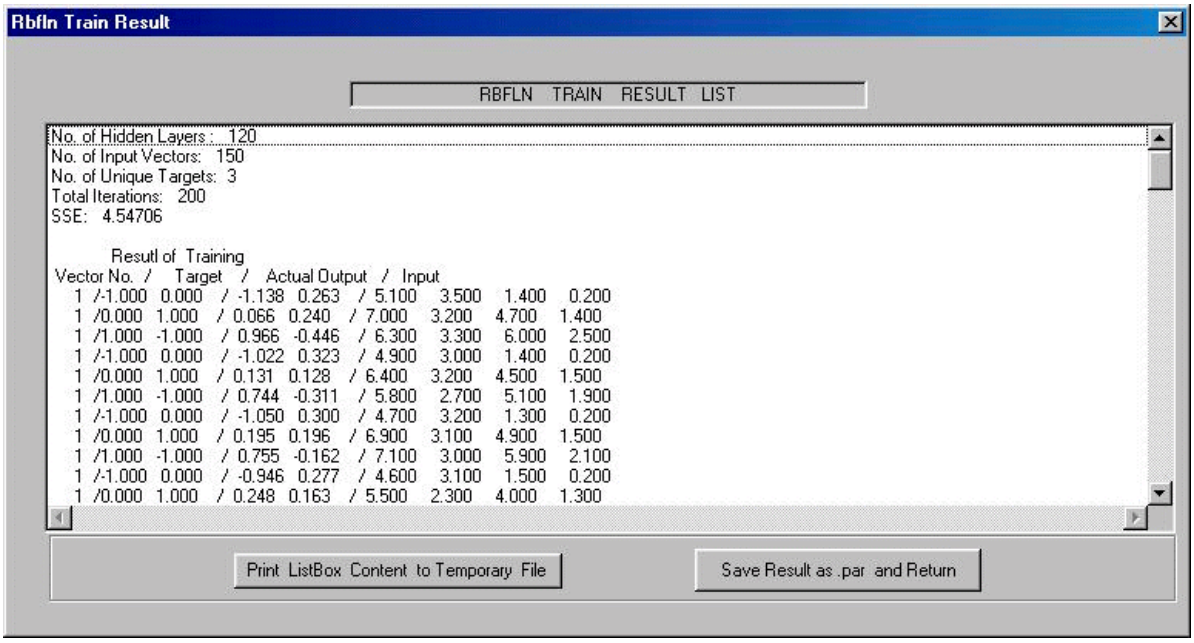


Fig. 16. The *RBFLN Train Result* dialog box.

4.3 RBFLN Test. Upon pressing the *Test* button in the *RBFLN* part of the *Main Work Interface* of Fig. 12, the user will get the same data file selection dialog box as in the *RBFLN Train* stage for selecting a data file to be used in the *RBFLN Test* stage (the default file suffix is also *.dta*). One can browse and select the training data file desired to use in the test and then select from another dialog box that comes up the appropriate parameters file used for the network being tested (the default suffix for parameter files is *.par*) as shown in Fig. 17 below. Here we see the *iris.par* parameter file.

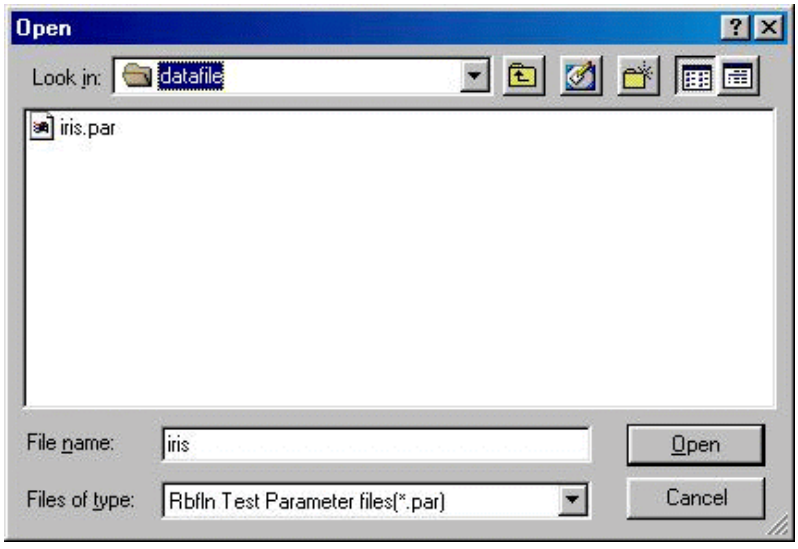


Fig. 17. Selecting a *.par* parameter file for *RBFLN Test*.

After the test procedure finishes, the *RBFLN Test Result* dialog box pops up to show the RBFLN network general parameters, the *SSE* of Equation (7) for evaluating the testing, and information as shown in Fig. 18. The user can print all mirrored content to a. *.txt* format file via the *Print ListBox Content to temporary file* button or return directly to the *Main Work Interface* via the *Return* button. These buttons are on the bottom of the dialog box of Fig. 18..

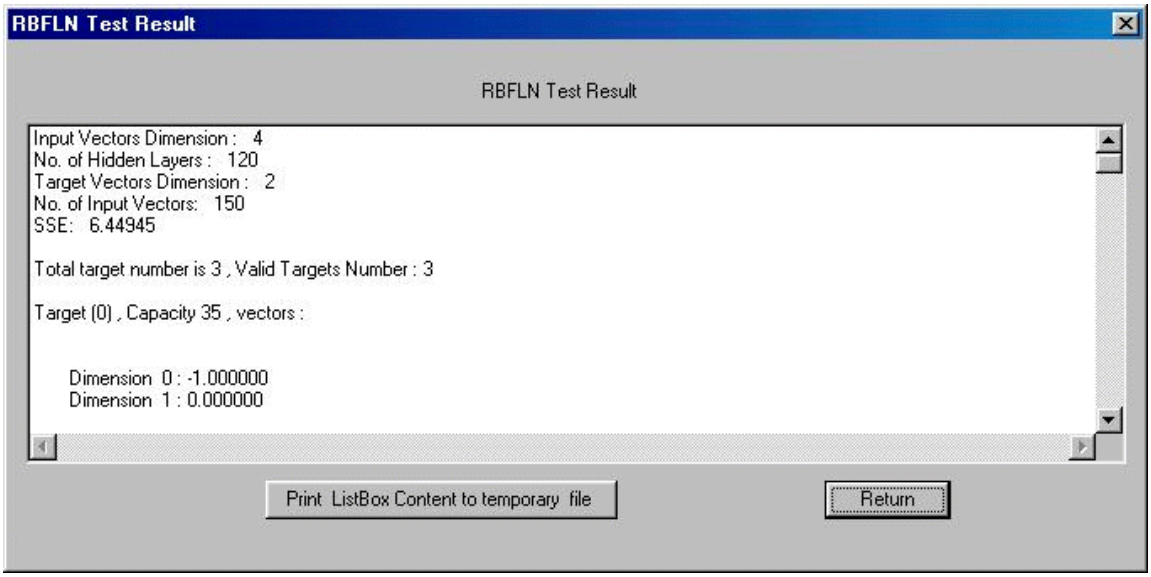


Fig. 18. RBFLN Test Result dialog box.

4.4 RBFLN Classification. When the user presses the *Classify* button of the *RBFLN* module in Figure 12, the same data file selection dialog box as we saw in the *RBFLN Train* function comes up (see Figure 13). One selects the data file to be used in the *RBFLN Classification* stage (the default file suffix is also *.dta*). The user can browse and select the desired data file. Then another dialog box comes up that is the same as the parameter file selection dialog box shown in the *RBFLN Test* function above. The purpose of this dialog box is for the user to select a parameter file to be used in the *RBFLN Classification* stage (the default file suffix is also *.par*). The user can browse and select the desired parameter file.

The classification procedure then begins. After it finishes, an *RBFLN Classify Result* dialog box as shown in Fig. 19 below prompts the user and shows the RBFLN network general parameters, *SSE, Total Targets No., Valid Target No.,* the *Capacity*, feature vector and target vector, and *Vector No./Classified Class No./Target Output/Actual Output*. The user can print all mirrored content in the dialog box to a *.txt* format file or save the classification results to a special RBFLN classification results file (with default suffix as *.rbn)*. The user can return to the *Main Work Interface* by clicking the *Return* button.

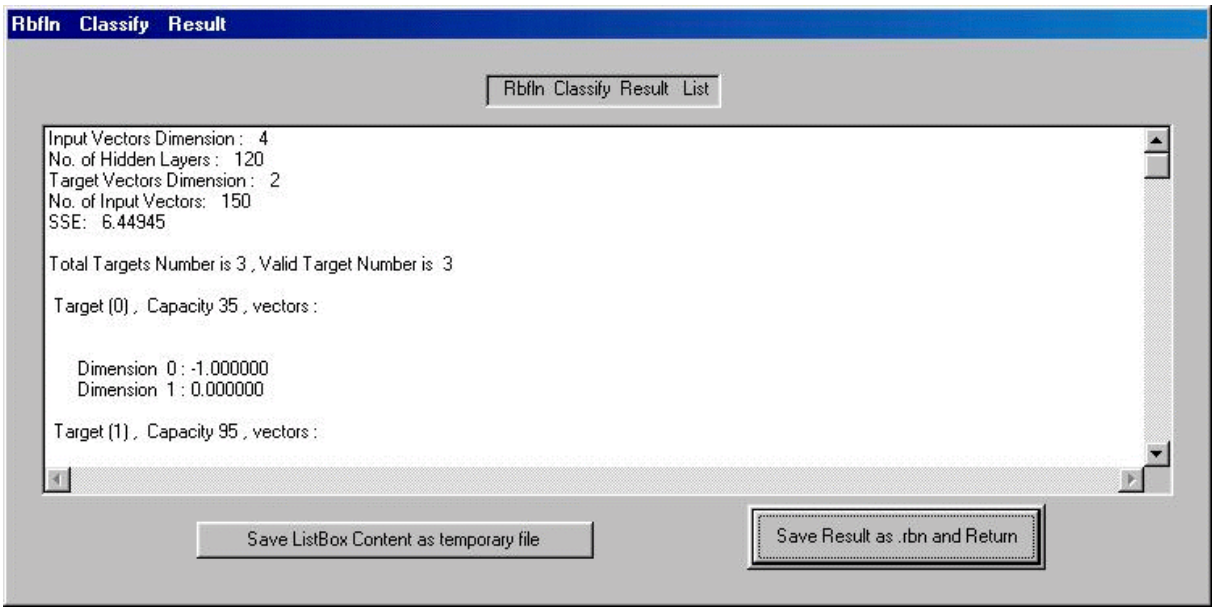


Fig. 19. The *RBFLN Classify Result* dialog box.

4.5 The Fuzzy (Unsupervised) Module. As shown in Fig. 12, the *FUZZY (Unsupervised)* module includes the two functions *Train* and *Classify*. *Train* uses unknown data (the data file may be the same as that used for the RBFLN training and may include target (labeled) outputs, but the targets are ignored). The fuzzy clustering finds the prototypes, or centers for K clusters. After gaining the optimal parameter set of these centers (saved as a *.cen* file), other unknown data files can be passed to the fuzzy clustering procedure based on this optimal parameter set. However, the fuzzy clustering procedure is a complete and independent self-organizing, unsupervised, learning algorithm and works very well on any set of unknown vectors by setting its own parameters. Thus it does not need any parameter files.

4.6 Fuzzy Training. Here one presses the *Train* button of the *FUZZY* module in Fig. 12 to get an OpenFile dialog box for the purpose of selecting a data file to be used in the *Fuzzy Train* stage (the default file suffix is *.dta* and the same files may be used as were used for the RBFLN training). Here we see the *iris.dta* file in Fig. 20 below.

The *Fuzzy Train Parameters* dialog box of Fig. 21 prompts the user by showing the content of current initialization parameter set for the *FUZZY* training procedure and it also enables the user to make custom modifications. The parameters include *Training Data File Name, No. Data Categories K, Closeness* and *Iterations* (the number of iterations desired). The default parameters are reasonable for essentially all cases. Also the training procedure is accumulative.

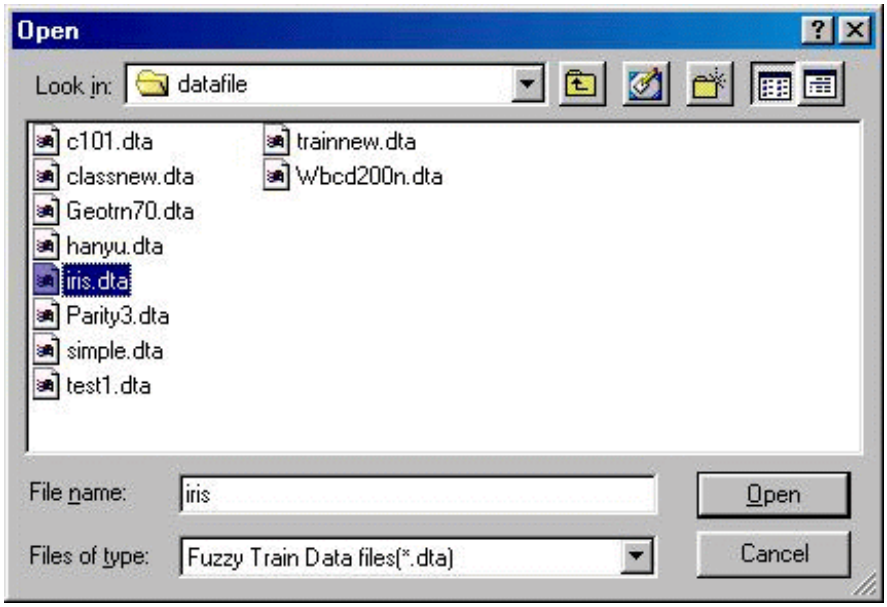


Fig. 20. Selecting a data file to be used in *Fuzzy Train*.

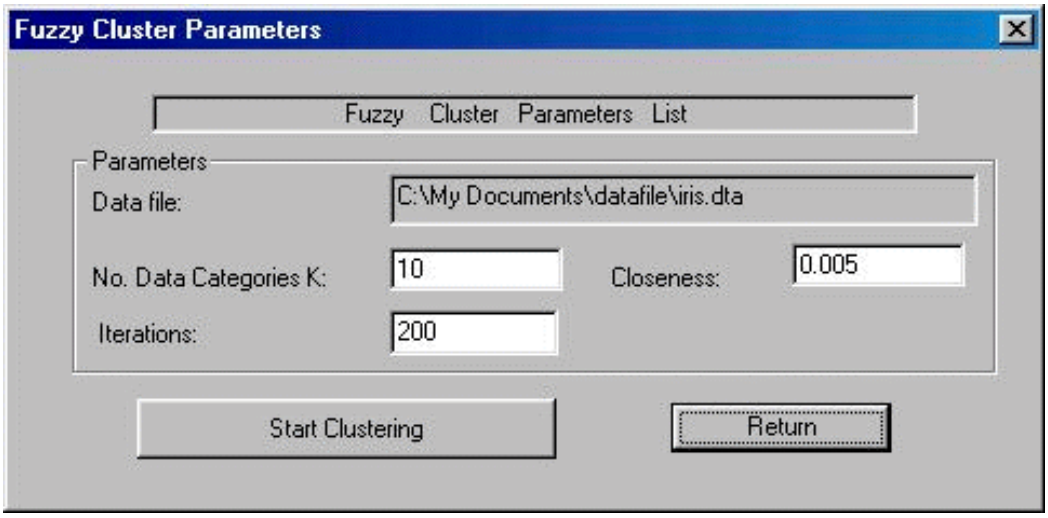


Fig. 21. The *Fuzzy Cluster Parameters* dialog box.

After clicking the *Start Clustering* button, the fuzzy clustering procedure starts. The *Fuzzy Train Processing Information* dialog box then prompts the user as shown in Fig. 22, which also presents the user with information as to which step of the fuzzy clustering algorithm is in process.

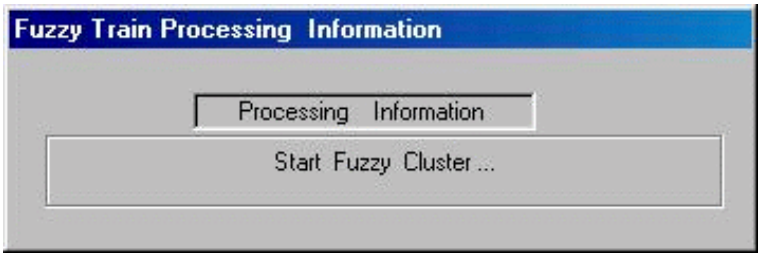


Fig. 22. Fuzzy Train Processing Information dialog box.

After the training procedure finishes, a Fuzzy Train Result dialog box comes up as in Fig. 23. It shows the No. of Classes, each center vector, the details of each class (including *Vectors No.*, *WFV* for the current class, and the indices for all vectors contained in each class), as well as the important clustering validity value (the larger it is, the better is the clustering) the reciprocal Xie-Bene measure.

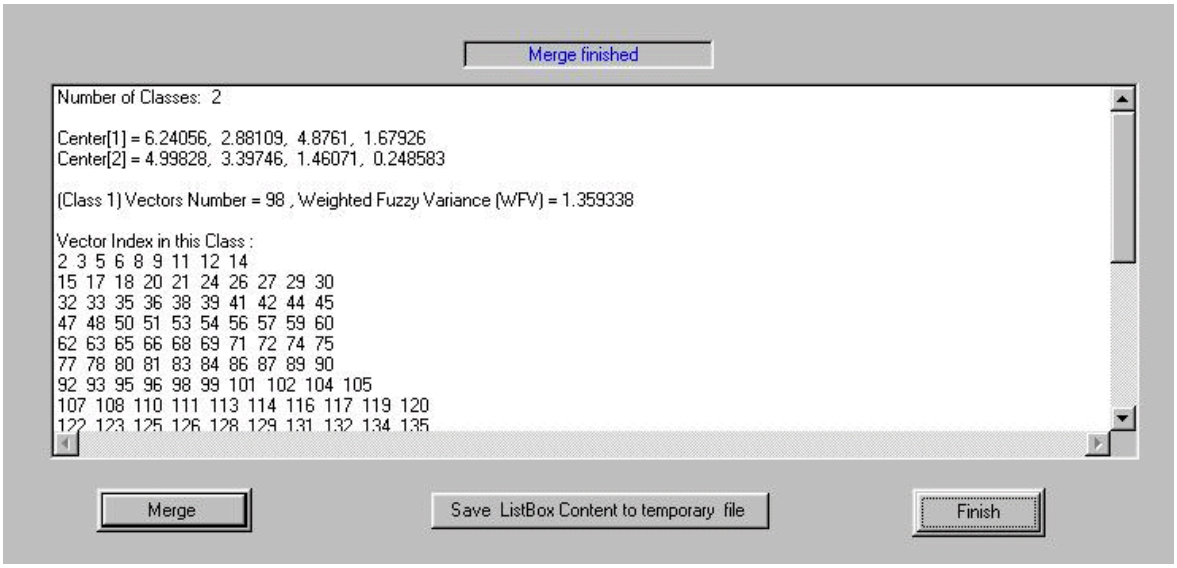


Fig. 23. The *Fuzzy Train Result* dialog box.

The user can now press the Merge button to further merge clusters that are close to each other. When the user is satisfied with the result, one can print all mirrored content of the dialog box to a *.txt* format file, or click the *Finish* button to save the current parameter set to a parameter file (the default suffix is *.cen*) for later classification use. Fig.24 shows the iris.cen parameter file as an example. Finally the user returns to the *Main Work Interface* to quit or do further processing. In the usual case, the training is the classification for fuzzy clustering, so any unknown data can be used to classify it merely by putting it through the fuzzy training as given above. Nothing further is required.

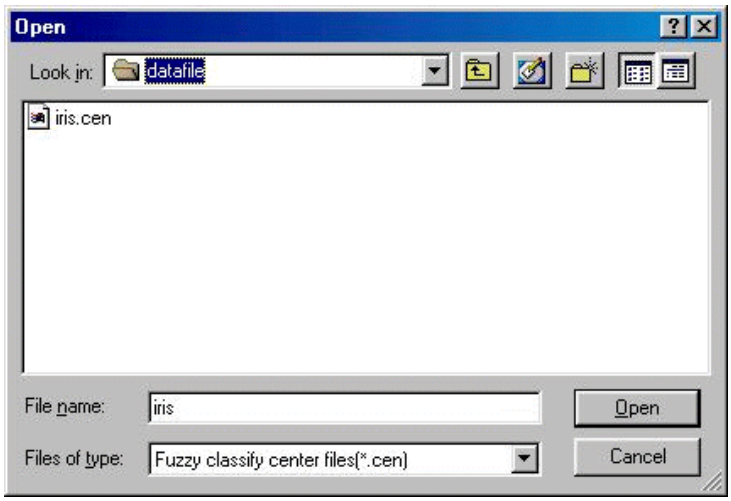


Fig. 24. Saving the *Fuzzy Train Result* parameter file.

4.7 Fuzzy Classification. In the process shown here, a trained fuzzy classifier is used to classify data other than that data which was trained on. This is not the usual or recommended process, but can be used in special cases. The user presses the *Classify* button in the *FUZZY* module in Fig. 12 (*Main Work Interface*) to get the same data file selection dialog box as in the *Fuzzy Train* function (the default file suffix is also *.dta*). The user can browse and select the desired data file for classification. The example we use is the *iris.dta* file. Another dialog box for selecting the parameter file saved from previous fuzzy clustering appears and prompts the user as shown in Fig. 25. Here one can browse and select the parameter file (default file affix is *.cen*) desired. We show the iris.cen as example that contains the centers from previous training.

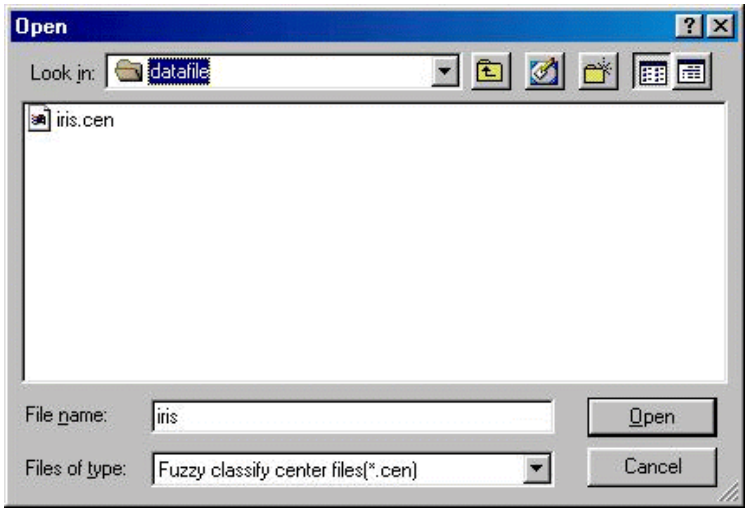


Fig. 25. Selecting a parameter file for further fuzzy classification.

After the classification procedure finishes, a *Fuzzy Classify Result* dialog box prompts the user as shown in Fig. 26. It also shows *No. of Classes*, each class center vector, details of each class (including *Vector No., WFV* for current class, and all vectors index contained in this class). It also shows the clustering validity (higher values are better). Clicking *Finish* returns to *Main Work Interface*.

The user can print all mirrored content in the dialog box to a *.txt* format file, or save the classification result to special *Fuzzy Classify Result* file (with default suffix *.fuz*). Fig. 27 shows the file *iris.fuz* as an example.

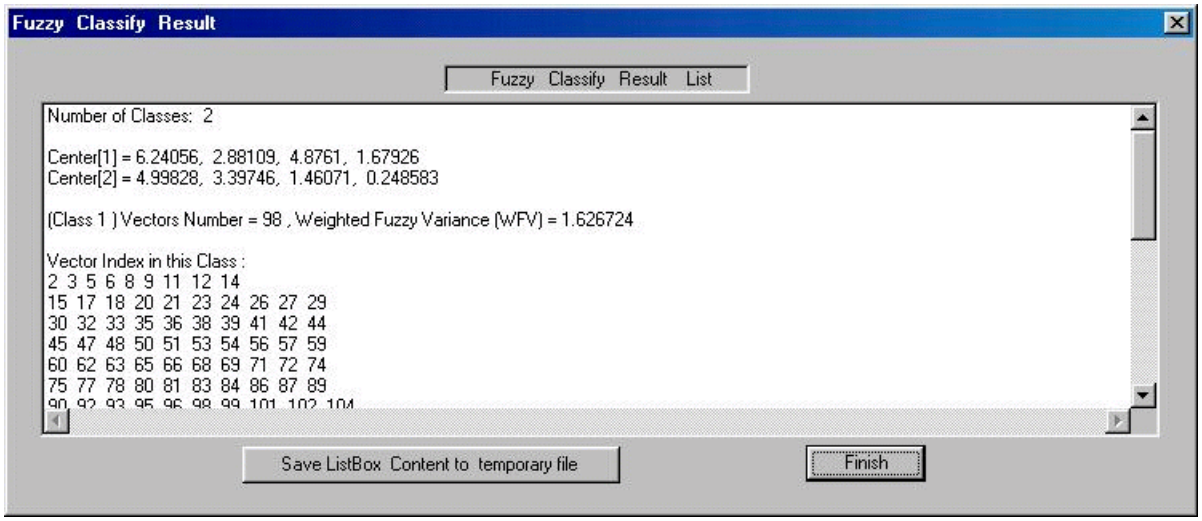


Fig. 26. The *Fuzzy Classify Result* dialog box.

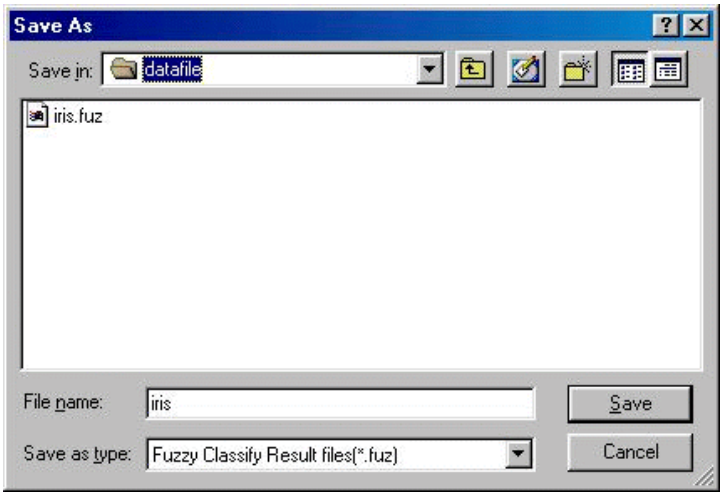


Fig. 27. Saving *Fuzzy Classification Result* data.

5. References

[1] E. Anderson, “The iris of the Gaspe peninsula,” *Bulletin American Iris Society*, vol. 59, 2-5, 1935.

[2] M. Bianchini, E. Frasconi, and M. Gori, "Learning without local minima in radial basis function networks," *IEEE Trans. on Neural Networks*, vol. 6, no. 3 (1995) 749-756.

[3] E. Forgy, “Cluster analysis of multivariate data: efficiency versus interpretability of classifications,” *Biometrics* 21, 768, 1965.

[4] E. J. Hartman, J. D. Keeler, and J. M. Kowalski, Layered neural networks with Gaussian hidden units as universal approximators, *Neural Computation*, vol. 2, no. 2 (1990) 210-215.

[5] K. Hornik, M. Stinchcombe, and H. White, Multilayer feedforward networks are universal approximators, *Neural Networks*, vol.2, no. 5 (1989) 359-366.

[6] C. T. Lin and C. S. George Lee, *Neural Fuzzy Systems*, Prentice-Hall, Upper Saddle River, NJ, 1995.

[7] Carl Looney, *Pattern Recognition Using Neural Networks*, Oxford University Press, N.Y., 1997.

[8] C. Looney, "Stabilization and speedup of convergence in training feedforward neural networks," *Neurocomputing* **10** (1996a) 7 - 31.

[9] C. Looney, "Demystifying feedforward neural networks as knowledge acquiring black boxes," *IEEE Trans. Knowledge and Data Engineering*, vol. 8, no. 2 (1996b) 211 - 226.

[10] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," *Proc. 5th Berkeley Symp. on Probability and Statistics*, University of California Press, Berkeley, 281-297, 1967.

[11] O. L. Mangasarian, W. N. Street and W. H. Wolberg, “Breast cancer diagnosis and prognosis via linear programming,” *Operations Research* vol. 43, no. 4 (1995) 570-577.

[12] N. R. Pal and J. C. Bezdek, “On cluster validity for the c-means model,” *IEEE Trans. on Fuzzy Systems*, vol. 3, no. 3, 1995.

[13]Y. H. Pao, G. H. Park, and D. J. Sobajic, Learning and generalization characteristics of the random vector functional link net, *Neurocomputing* **6** (1994) 163 - 180.

[14] J. Park and I. W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural Computation*, vol. 3 (1991) 246-257.

[15] M. Schneider and M. Craig, "On the use of fuzzy sets in histogram equalization," *Fuzzy Sets Syst.*, vol. 45, 271-278, 1992.

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

[17] W. H. Wolberg and O. L. Mangasarian, "Multisurface method of pattern separation for medical diagnosis applied to breast cytology," Proc. of the Nat. Academy of Sciences, U.S.A., Volume 87, 9193-9196, December, 1990.

[18] W. H. Wolberg, W. N. Street, D. M. Heisey and O. L. Mangasarian, "Computer derived nuclear features distinguish malignant from benign breast cytology," *Human Pathology* 26 (1995) 792-796.

[19] X. L. Xie and G. Beni, "A validity measure for fuzzy clustering," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 13, no. 8, 841-847, 1991.