Neural Networks 21 (2008) 1537–1547

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

A fast nearest neighbor classifier based on self-organizing incremental neural network

Shen Furao [a,*], Osamu Hasegawa [b]

[a] The State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210093, PR China

[b] Imaging Science and Engineering Lab., Tokyo Institute of Technology, Japan

ARTICLE INFO

Article history:
Received 29 December 2007
Received in revised form
21 May 2008
Accepted 2 July 2008

Keywords:
Self-organizing incremental neural network
Nearest neighbor
Fast
Prototype-based classifier

ABSTRACT

A fast prototype-based nearest neighbor classifier is introduced. The proposed Adjusted SOINN Classifier (ASC) is based on SOINN (self-organizing incremental neural network), it automatically learns the number of prototypes needed to determine the decision boundary, and learns new information without destroying old learned information. It is robust to noisy training data, and it realizes very fast classification. In the experiment, we use some artificial datasets and real-world datasets to illustrate ASC. We also compare ASC with other prototype-based classifiers with regard to its classification error, compression ratio, and speed up ratio. The results show that ASC has the best performance and it is a very efficient classifier.

1. Introduction

k-nearest neighbors algorithm (Cover & Hart, 1967) is very useful for some applications such as machine learning, data mining, natural language understanding, and information retrieval (Dasarathy, 1991). Let $T = \{t_i \in \omega, i = 1, 2, \ldots, m\}$ denotes a set of training patterns, each pattern $t_i \in T$ has a class label. The target of kNN is to find the k-nearest neighbors of a test pattern x ($x \in \omega$) in T based on a dissimilarity measure $d(.,.)$, and then classify the pattern x with the same label as the majority voting of nearest patterns in the training set. We call this classifier as Nearest Neighbor Classifier (NNC(k)) (Cover & Hart, 1967). If we set $k = 1$, the k-nearest neighbors classifier becomes 1-nearest neighbor (1-NN) classifier.

The main advantages of kNN include that it can learn from a small set of examples, can incrementally add new information at run time, no optimization required, capable to model very complex target functions by a collection of less complex approximations, etc.; on the other hand, its major disadvantage is that it is computationally intensive for large datasets.

kNN uses all training data as the prototypes. To reduce the amount of required storage and improve the classification speed, we need to reduce the number of prototypes. The question becomes, for the training dataset $T = \{t_i \in \omega, i = 1, 2, \ldots, m\}$, to find a set P with M prototypes that represent T such that P can be used for classification using the nearest neighbor rule.

For the prototype-based algorithms, the question is how one knows when there are enough prototypes and how to prevent overfitting to training data. NNC uses all the training data to label unseen patterns. The Nearest Mean Classifier (NMC) (Hastie, Tibshirani, & Friedman, 2001) only stores the mean of each class, i.e. one prototype per class. It generally has a high error on the training and test data. There are lots of other prototype-based algorithms such as k-means classifier (KMC) (Hastie et al., 2001), Learning Vector Quantization (LVQ) (Kohonen, 1990), and others (Bezdek & Kuncheva, 2001; Wilson & Martinez, 2000). Such methods select a fixed number of prototypes per class as an overfitting avoidance strategy. But when the class distributions differ from each other, either in the number of patterns, the density of the patterns, or the shape of the classes, the optimal number of prototypes may be different for each other. Nearest Subclass Classifier (NSC) (Veenman & Reinders, 2005) tried to impose the number of prototypes per class by introducing a variance constraint parameter. They assume that the features of every pattern contain the same amount of noise and do not model label noise, and they assume the undersampling of the classes is the same everywhere in feature space. Such assumptions may not be satisfied in real world task.

For some prototype-based classifiers, it is very difficult to incrementally add new information at run time, and to eliminate the influence of noise. Here, noise means the unknown amount of noise in the features and class labels of the training dataset.

In this paper, we propose a prototype-based nearest neighbor method that is based on the self-organizing incremental neural

* Corresponding author. Tel.: +81 45 924 5180; fax: +81 45 924 5175.
E-mail address: frshen@nju.edu.cn (F. Shen).

0893-6080/$ – see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.neunet.2008.07.001

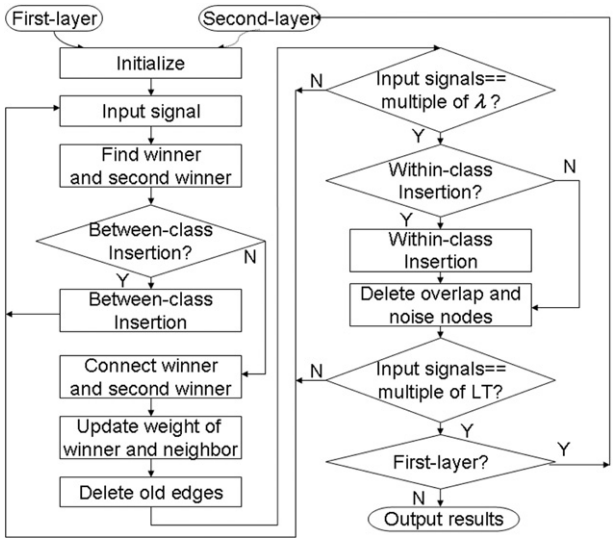

Fig. 1. Learning process of SOINN.

network (SOINN) (Shen & Hasegawa, 2006, 2005). The goals of the proposed method are: (1) automatically learn the number of prototypes needed to represent every class, if needed, allocate different number of prototypes for different classes with different distribution or shape; (2) learn new information without destroying old learned information, i.e., to realize incremental learning; (3) reduce prototypes caused by noise in order to decrease the misclassification, i.e., the proposed method must be robust to noisy training data; (4) delete unnecessary prototypes during the classification process to accelerate the classification speed, i.e. only the prototypes used to determine the decision boundary will be remained.

During the classification process, if not stated differently, for all experiments in this paper we will employ the 1-nearest neighbor (prototype) rule to classify patterns based on the generated set of labeled prototypes.

2. Overview of self-organizing incremental neural network (SOINN)

In this section, we introduce the self-organizing incremental neural network (SOINN) (Shen & Hasegawa, 2006), which is the basis of the proposed method. SOINN adopts two-layer network. The training results of first-layer will be used as the training set for second-layer. The targets of SOINN are realizing the unsupervised learning and represent the topology structure of input distribution. We summarize the flowchart of SOINN in Fig. 1.

When an input vector is given to SOINN, it finds the nearest node (winner) and the second nearest node (second winner) of the input vector. It subsequently judges if the input vector belongs to the same cluster of the winner or second winner using the similarity threshold criterion. The similarity threshold T_i is defined as the distance (Euclidean distance) from the boundary to the center of Voronoi region V_i of node i. During the learning process, the node i will change its position to meet the inputting pattern distribution, and thus the Voronoi region V_i of the node i will also change, therefore, the similarity threshold T_i will also change.

Algorithm 2.1. Calculation of similarity threshold T

(1) Initialize the similarity threshold of node i to $+\infty$ when node i is generated as a new node.
(2) When node i is a *winner* or *second winner*, update similarity threshold T_i:

- If the node has direct topological neighbors, T_i is updated as the maximum distance between node i and all of its neighbors,
$$T_i = \max_{c \in N_i} \|\boldsymbol{W}_i - \boldsymbol{W}_c\|, \quad (1)$$
here, N_i is the neighbor set of node i.
- If node i has no neighbor, T_i is updated as the minimum distance of node i and all other nodes in A,
$$T_i = \min_{c \in A \setminus \{i\}} \|\boldsymbol{W}_i - \boldsymbol{W}_c\| \quad (2)$$
here, A is the node set.

Here, *winner* means the nearest node to the input pattern, and *second winner* means the second nearest node to the input pattern. $\boldsymbol{W}_i$ is the weight vector of node i.

The input vector will be inserted to the network as a new node to represent the first node of a new class if the distance between the input vector and the winner or second winner is greater than the similarity threshold of a winner or second winner. This insertion is called a between-class insertion because this insertion will result in the generating of a new class, even if the generated new class might be classified to some older class in the future.

If the input vector is judged as belonging to the same cluster of winner or second winner, and if no edge connects the winner and second winner, connect the winner and second winner with an edge, and set the 'age' of the edge as '0'; subsequently, increase the age of all edges linked to the winner by '1'.

Then, update the weight vector of the winner and its neighboring nodes. We use i to mark the winner node, and M_i to show the times for node i to be a winner. The change to the weight of winner $\Delta \boldsymbol{W}_i$ and change to the weight of the neighbor node $j(\in N_i)$ of $i\Delta \boldsymbol{W}_j$ are defined as

$$\Delta \boldsymbol{W}_i = \frac{1}{M_i}(\boldsymbol{W}_s - \boldsymbol{W}_i) \quad (3)$$

and

$$\Delta \boldsymbol{W}_j = \frac{1}{100 M_i}(\boldsymbol{W}_s - \boldsymbol{W}_j) \quad (4)$$

where $\boldsymbol{W}_s$ is the weight of the input vector.

If the age of one edge is greater than a predefined parameter a_d, then remove that edge.

After λ learning iterations, the SOINN inserts new nodes into the position where the accumulating error is extremely large. Cancel the insertion if the insertion cannot decrease the error. The insertion here is called within-class insertion because the new inserted node is within the existing class; also, no new class will be generated during the insertion. Then SOINN finds the nodes whose neighbor is less than or equal to 1 and deletes such nodes based on the presumption that such nodes lie in the low-density area.

After LT learning iterations of the first layer, the learning results are used as the input for the second layer. The second layer of SOINN uses the same learning algorithm as the first layer.

SOINN adopts two schemes to insert new nodes and thus realize the incremental learning and topology representation: between-class insertion and within-class insertion. In order to realize the within-class insertion, SOINN uses 5 user determine parameters and another parameter "error-radius" to judge if the insertion is successful, which makes the system complicated and difficult to understand.

In fact, during the training of first-layer of SOINN, between-class insertion is the main part, and within-class insertion has little contribution for inserting new nodes. During the training of second-layer of SOINN, both between-class insertion and within-class insertion are needed to make the number of nodes enough for representing topology structure (Shen & Hasegawa, 2007).

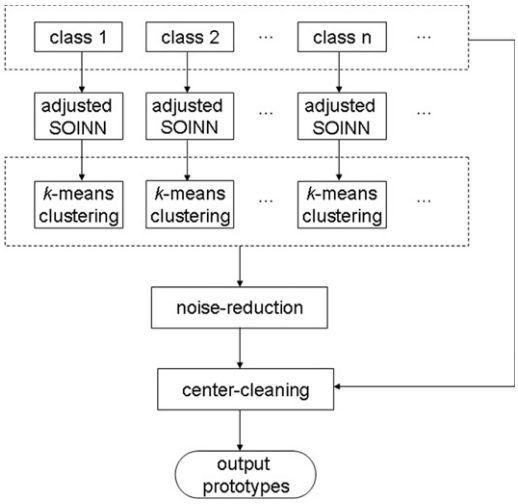

Fig. 2. Learning process of adjusted SOINN classifier (ASC).

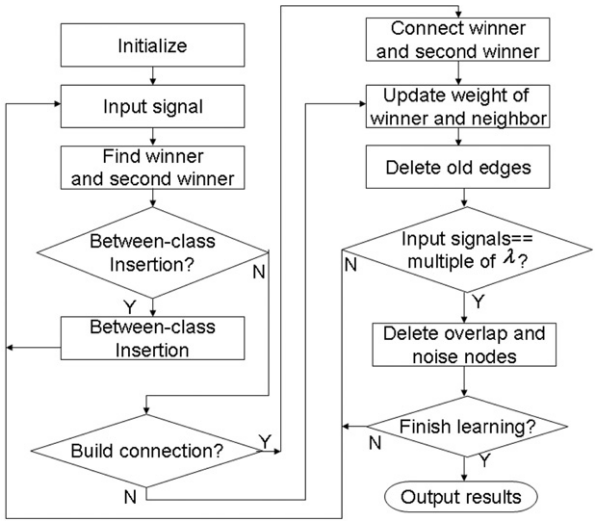

Fig. 3. Flowchart of adjusted SOINN.

3. Adjusted SOINN classifier (ASC)

The proposed Adjusted SOINN Classifier (ASC) inherited some properties of SOINN such as incremental learning, robust to noise, and automatically learn number of prototypes needed to represent every class. The shortcoming of SOINN is that it needs too much parameters to realize the within-class insertion; it is not stable and the results depend on the input sequence of the training data; also, the target of SOINN is to realize unsupervised learning and topology representation, and here we want to do supervised learning and use as less prototypes as possible to realize fast classification. We improve SOINN with the following aspects: (1) adjust SOINN with less parameters to represent the topology structure of input data; (2) improve SOINN to get more stable results with the help of *k*-means clustering (Hastie et al., 2001); (3) design the noise-reduction part to reduce some prototypes caused by noise; and (4) design the center-cleaning part to delete those unnecessary prototypes during the classification process. Because the proposed method is based on SOINN, we name the proposed algorithm Adjusted SOINN Classifier (ASC). The flowchart of ASC is shown in Fig. 2.

From Fig. 2 we know that, at first, ASC does adjusted SOINN for every class separately, then does *k*-means clustering with the results of adjusted SOINN for every class, then use all the *k*-means results to do noise-reduction. At last, ASC uses the input data of all classes and the results of noise-reduction part to do center-cleaning process.

3.1. Adjusted SOINN

As we pointed out in Section 2, during the training of first-layer of SOINN, between-class insertion is the main part, and within-class insertion has little contribution for inserting new nodes. The target of second-layer is to delete redundant nodes, separate overlapped clusters, and delete nodes caused by noise. Just for the topology representation target, the first-layer can get better topology representation than second-layer (Shen & Hasegawa, 2007). Here we only adopt the first-layer of SOINN as the basis of the proposed ASC method, and delete the within-class insertion part to make it easy to understand and save 5 user-determine-parameters. The deletion of within-class insertion will not influence the learning results. It is because if we only adopt single-layer network, the between-class insertion assures that the density of nodes will be enough to represent the topology structure. With the definition of similarity threshold, inserted new nodes may come from not happened area, and the following process (such as competitive Hebbian rule) will link such new nodes with old nodes. The adaptively updated similarity threshold will not be too large to make the nodes sparse. With between-class insertion, when the nodes reach the boundary of a class, the insertion will be automatically stopped for the class, and we avoid the permanent increase of nodes. Fig. 3 gives the flowchart of adjusted SOINN.

When an input vector is given to adjusted SOINN, it finds the winner and second winner of the input vector, then judges whether the input vector belongs to the same cluster of the winner or second winner using the criterion of similarity threshold. The similarity threshold T_i is calculated using Algorithm 2.1.

If the distance between the input vector and the winner or second winner is greater than the similarity threshold of winner or second winner, the input vector will be inserted to the network with between-class insertion process.

If the input vector is judged as belonging to the same cluster of winner or second winner, we update the weight vector of winner and its neighbor nodes with formula (3) and (4).

If no edge connects the winner and second winner, we connect the winner and second winner with an edge, and set the 'age' of the edge as '0'; subsequently, we increase the age of all edges linked to the winner by '1'. If the age of one edge is greater than a predefined parameter a_d, then we remove that edge.

After λ learning iterations, adjusted SOINN finds the nodes whose neighbor is less than or equal to 1 and deletes such nodes. Algorithm 3.1 is the detail algorithm of adjusted SOINN.

Algorithm 3.1. Adjusted SOINN

(1) Initialize node set A to contain two nodes, c_1 and c_2 with weight vectors chosen randomly from the input pattern. Initialize connection set C, $C \subset A \times A$, to the empty set.
(2) Input new pattern $\xi \in R^n$.
(3) Search the nearest node (*winner*) s_1, and the second-nearest node (*second winner*) s_2 by

$$s_1 = \arg\min_{c \in A} \|\xi - \boldsymbol{W}_c\| \tag{5}$$

$$s_2 = \arg\min_{c \in A \setminus \{s_1\}} \|\xi - \boldsymbol{W}_c\|. \tag{6}$$

If the distance between ξ and s_1 or s_2 is greater than similarity threshold T_{s_1} or T_{s_2}, the input signal is a new node, add it to A and go to step (2) to process the next signal. The similarity threshold T is calculated by Algorithm 2.1.

(4) If a connection between s_1 and s_2 does not exist, create it. Set the age of the connection between s_1 and s_2 to zero.
(5) Increment the age of all edges emanating from s_1 by 1.
(6) Adapt the weight vectors of the *winner* and its direct topological neighbors by fraction $\epsilon_1(t)$ and $\epsilon_2(t)$ of the total distance to the input signal,

$$\Delta \boldsymbol{W}_{s_1} = \epsilon_1(t)(\xi - \boldsymbol{W}_{s_1}) \tag{7}$$

and for all direct neighbors i of s_1,

$$\Delta \boldsymbol{W}_i = \epsilon_2(t)(\xi - \boldsymbol{W}_i). \tag{8}$$

We adopt a scheme to adapt the learning rate over time by

$$\epsilon_1(t) = \frac{1}{t} \tag{9}$$

$$\epsilon_2(t) = \frac{1}{100t}. \tag{10}$$

(7) Remove edges with an age greater than a predefined threshold a_d. If this results in nodes having no more emanating edges, remove them as well.
(8) If the number of input signals generated so far is an integer multiple of parameter λ, delete some nodes as follows: for all nodes in A, search for nodes having no neighbor or only one neighbor, then remove them.
(9) Go to Step (2) to continue the learning until the learning time is satisfied.

In Algorithm 3.1, we need to determine two parameters a_d and λ. These two parameters will influence the frequency of deleting connections between nodes and nodes lie in sparse area. Thus, if we want to save previous learned knowledge much longer, we choose large value for these two parameters, and get lots of nodes to realize low classification error; on the other hand, if we want less nodes to save memory space and speed up the classification, we set the value of these two parameters small to remove nodes and edges frequently. It means that, the two parameters are depending on the real condition of the task, we can use these parameters to control the recognition performance of ASC.

3.2. k-means clustering

k-means clustering (Hastie et al., 2001) is a method for finding clusters and cluster centers in a set of unlabeled data. We choose the desired number of cluster centers m, give an initial set of centers $c_j(0), j = 1, \ldots, m$, the k-means algorithm alternates the two steps: (1) for each center we identify the subset of training points (its cluster) that is closer to it than any other center; (2) the means of each feature for the data points in each cluster are computed, and this mean vector becomes the new center for that cluster. These two steps are iterated until convergence.

Algorithm 3.2. k-means clustering

(1) Partition the inputting vector data $x_i, i = 1, \ldots, n$ into the channel symbols using the minimum distance rule. This partitioning is stored in a $n \times m$ indicator matrix S whose elements are defined by

$$s_{ij} = \begin{cases} 1 & \text{if } d(x_i, c_j(k)) = \min\limits_p d(x_i, c_p(k)) \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

(2) Determine the centroids of the inputting data by channel symbol. Replace the old centers with these centroids:

$$c_j(k+1) = \frac{\sum\limits_{i=1}^{n} s_{ij} x_i}{\sum\limits_{i=1}^{n} s_{ij}}, \quad j = 1, \ldots, m. \tag{12}$$

(3) Repeat step (1)–step (2) until no $c_j, j = 1, \ldots, m$ changes anymore.

k-means heavily depends on the initial value of centers $c_j(0), j = 1, \ldots, m$. The difficulties of k-means are how to determine the number of centers m in advance and how to find good initial value of centers. To solve such difficulties, in ASC, we use the learned number of nodes of adjusted SOINN as the number of centers, and use the position (weight vector) of adjusted SOINN nodes as the initial value of centers.

In ASC, adjusted SOINN is not stable and the results depend on the sequence of input data. The number of nodes and the position of nodes are different if we repeat the training under same environment with different input sequence. With the help of k-means clustering, we move such nodes to the center of the clusters and improve the stability of ASC. It is because that the generated nodes of adjusted SOINN represent the topology of input data, such nodes are distributed near the centers of sub-clusters, and k-means clustering moves such nodes to the centers of sub-clusters.

3.3. Noise-reduction part

If there is noise in the training data, during the training of adjusted SOINN, some nodes will be generated by noise. We only adopt single-layer in adjusted SOINN. Therefore, we cannot remove all nodes generated by noise if there is plenty of noise. Here, noise means that the training dataset contains an unknown amount of noise in the features and class labels. To prevent the nearest neighbor rule from fitting to the noisy training data without restriction, we use the idea of an early method: k-Edited Neighbors Classifier (ENC) (Wilson, 1972) in ASC, i.e. if the label of a node differs from the label of majority voting of its k-neighbors, it is considered an outlier and the node is removed from the set of prototypes. Algorithm 3.3 is the detailed algorithm of the Noise-reduction part.

Algorithm 3.3. Noise-reduction

(1) For a prototype c in prototype set C, find k nearest prototypes of prototype c.
(2) Delete prototype c from the prototype set C if the major voting of k nearest prototypes has a different class label with prototype c.
(3) Repeat this process until all prototypes are processed.

In this noise-reduction part, we can use some methods such as cross-validation to tune the parameter k.

3.4. Center-cleaning part

The prototype set obtained using adjusted SOINN and k-means clustering can be used to represent the topology structure of the input data. However, during the classification process, some prototypes in the central part of a class might not be useful because, during the classification process, we use the one-nearest neighbor (prototype) rule to classify patterns based on the generated set of labeled prototypes, and only prototypes lie in the boundary can be used. We must delete those prototypes which lie in the central parts of classes to save memory space of storage for prototypes and to accelerate the classification speed. We designed Algorithm 3.4 to realize this target. Algorithm 3.4 is based on this idea: if a prototype of a class has never been the nearest prototype to other classes, the prototype lies in the central part of the class. Therefore, we delete it.

Algorithm 3.4. Center-cleaning process

(1) Suppose that there are n classes. Given class i, $i = 1, \ldots, n$, do the following steps.
(2) For all samples of other classes that differ from class i, find the nearest prototype of class i generated by adjusted SOINN and k-means clustering.

(3) Delete this prototype if a prototype of class i has never been the nearest prototype of other classes. Repeat this step until no prototype can be deleted.

After executing of Algorithm 3.4, the remaining prototypes are all useful prototypes for the one-nearest-neighbor rule. Removed prototypes according to the center-cleaning process have no usage for the classification process. We must mention that the cleaning rule is based on the training set; it is possible for this process to delete some useful prototypes for the testing set and leads to a loss of the recognition performance, but if the testing set obeys the same distribution as the training set, the removal will not decrease the efficiency and can accelerate the speed greatly.

3.5. ASC algorithm

With the analysis of Sections 3.2–3.4, we give the whole learning algorithm of adjusted SOINN classifier (ASC) in Algorithm 3.5.

Algorithm 3.5. Learning algorithm of ASC

(1) Suppose there are n classes, for every class, do adjusted SOINN (Algorithm 3.1), Algorithm 3.1 outputs the number of prototypes $\{N_i, i = 1, \ldots, n\}$ of every class, and give the weight vector of such prototypes $\{AS(c_j, i), j = 1, \ldots, N_i; i = 1, \ldots, n\}$.
(2) For every class, do k-means clustering (Algorithm 3.2). The number of centers for every class may be different, we adopts the N_i generated by step (1) as the number of centers for class i. The $\{AS(c_j, i), j = 1, \ldots, N_i\}$ will be the initial value of centers for class i. The results of this step will be $\{km(c_j, i), j = 1, \ldots, N_i; i = 1, \ldots, n\}$.
(3) For all prototypes in $\{km(c_j, i), j = 1, \ldots, N_i; i = 1, \ldots, n\}$, use Algorithm 3.3 to remove those prototypes caused by noise (noise reduction), and we get $\{NR(c_j, i), j = 1, \ldots, \hat{N}_i; i = 1, \ldots, n\}$, here $\hat{N}_i$ is the new number of prototypes of class i, it will be equal to or less than N_i.
(4) For $\{NR(c_j, i), j = 1, \ldots, \hat{N}_i; i = 1, \ldots, n\}$, with all samples of all classes, use Algorithm 3.4 to delete central prototypes of every class (center cleaning), which are not useful for classification processes, and we get final prototype set $\{Prototype(c_j, i), j = 1, \ldots, M_i; i = 1, \ldots, n\}$, M_i is the final number of prototypes of every class, it is equal to or less than $\hat{N}_i$, and $\sum_{i=1}^{n} M_i$ is the final total number of prototypes. Such prototypes will be used for classification of new objects.

In Algorithm 3.5, first, we execute the adjusted SOINN separately for every class; then we use the training results of adjusted SOINN (number of prototypes and the position of prototypes) to do k-means clustering; then we use the k-means results of all classes to reduce some prototypes caused by noise. Finally, we use the remaining prototypes and all training data to find unnecessary prototypes that lie in the central part of node distribution, then remove them to speed up the process.

In the above algorithm, there are two parameters (a_d, λ) needed by adjusted SOINN, and we also need one parameter k in step (3) to realize the noise-reduction process. We have given the discussion of such parameters in Sections 3.2 and 3.3.

For the prototype-based classifier, the classification error, storage requirements, and speed can be used to evaluate the performance of a classifier. In this paper, we measure the memory requirements with the compression ratio r_c of the trained classifier, and measure the comparison of classification speed with the speed up ratio $r_s(A, B)$ between trained classifiers A and B, where

$$r_c = \frac{\text{number of prototypes}}{\text{train data size}} \tag{13}$$

$$r_s(A, B) = \frac{\text{classification speed of classifier } A}{\text{classification speed of classifier } B} \tag{14}$$

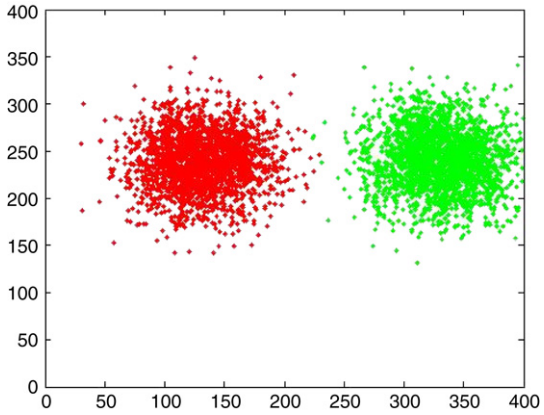

Fig. 4. Two Gaussian distribution datasets, without overlap (2 classes).

and use classification error, compression ratio, and speed up ratio to validate the performance of classifiers.

4. Experiment

In this part, at first, we use some 2-dimension artificial datasets to test the ASC and illustrator the detail of ASC; then through the test on a real-word dataset we prove the efficiency of the proposed method. At last we compare ASC with some other typical prototype-based classifiers.

In the following experiment, for software environment, the operating system is WinXP and the programming language is Visual C++6.0. Hardware uses a PC with an Intel Xeon(TM) CPU 3.20 GHz and 2.0 GB RAM.

4.1. Artificial dataset

During the test of all artificial datasets, we set the parameters as following: $a_d = 20$, $\lambda = 20$, $k = 5$. The a_d and λ will influence the number of generated prototypes, i.e. compression ratio, but they will not influence the recognition performance if we can get enough prototypes. The parameter k is not sensitive, we can choose other value for it and get the same recognition results. For all experiments in this section, we do 10 times learning and testing, and then give the average classification error and compression ratio as the recognition performance.

(1) Experiment 1: two Gaussian distribution without overlap

In this experiment, we adopt two non-overlapped Gaussian distribution (as shown in Fig. 4). For every class, we choose 2000 samples as the training pattern, and choose 200 samples as the test pattern. It means there 4000 training samples and 400 test samples. The classification error of 1-NN classifier for this example is 0.0%.

Then we train the ASC with Algorithm 3.5. Fig. 5 gives the adjusted SOINN result, it is nearly the same as the results of first-layer of SOINN. It shows that adjusted SOINN can represent the topology of two classes well. Fig. 6 is the training results of ASC. ASC removed lots of prototypes generated by adjusted SOINN, such removed prototypes are not useful for the following classification. It shows that to realize the classification, ASC only need 6 prototypes (3 for one class, and 3 for another class). With these 6 prototypes, we classify the test set and get 0.0% classification error. Compared with 1-NN classifier, ASC uses $6/4000 = 0.15\%$ (compression ratio $r_c = 0.15\%$) prototypes of 1-NN and get the same classification error.

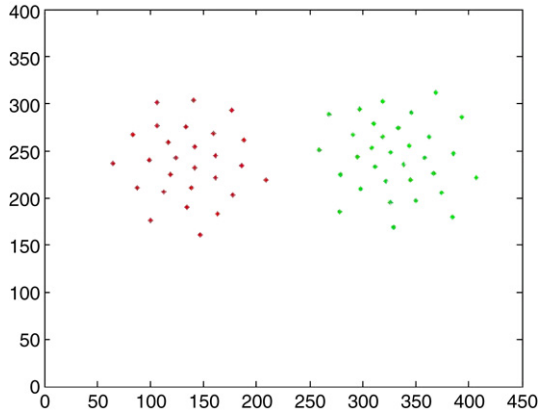

Fig. 5. Adjusted SOINN results of Fig. 4, it is nearly the same as the results of first-layer of SOINN.

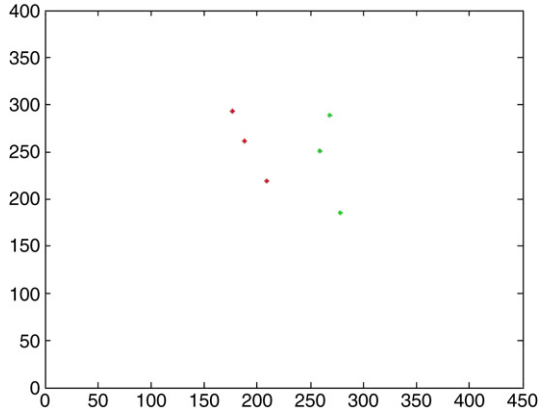

Fig. 6. ASC results of Fig. 4.

Fig. 7. Two Gaussian distribution, two concentric rings, and sinusoid curve, without overlap (5 classes).

(2) Experiment 2: five classes with different distribution and different shape, without overlap

In this experiment, we add three other classes to the classes in Experiment 1. The classes obey the distribution shown in Fig. 7. They are two Gaussian distribution, two concentric rings, and one sinusoid curve. We randomly took 2000 samples from every class as the training pattern, and randomly took 200 samples from every class as the test pattern, i.e. there are 10,000 samples in training set and 1000 samples in test set. The classification error of 1-NN is 0.0%.

Fig. 8 is the adjusted SOINN results. It represents the topology structure of input data very well. Fig. 9 is the ASC results. For the two Gaussian distribution classes, the number of prototypes becomes larger than in Experiment 1, it is because there are other classes outside the two classes, and to build decision boundary between the Gaussian distribution classes and other classes, it needs more prototypes. For the concentric rings and sinusoid, ASC also automatically determined the number of prototypes needed to decide the decision boundary. Fig. 9 also shows that for different classes, ASC allocates different number of prototypes for the reason that different classes obey different distribution or have different shape and size. The classification error of ASC is 0.0% and it needs 69 prototypes, i.e., ASC uses 69/10,000 = 0.69% (r_c = 0.69%) prototypes of 1-NN and get the same classification error.

Fig. 8. Adjusted SOINN results of Fig. 7, it is nearly the same as the results of first-layer of SOINN.

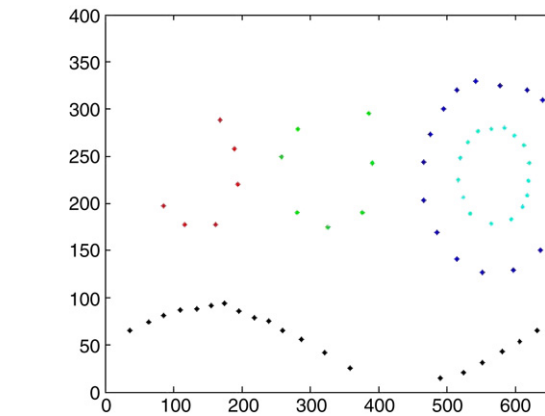

Fig. 9. ASC results of Fig. 7.

(3) Experiment 3: five classes with different distribution and different shape, with overlap

In this experiment, we adjust the classes in Experiment 2 by move two Gaussian distribution classes together to form overlapped classes. Fig. 10 is the distribution. We randomly took 2000 samples from every class as the training pattern, and randomly took 200 samples from every class as the test pattern. The classification error of 1-NN is 2.7%.

Fig. 11 is the adjusted SOINN results, and there are overlap prototypes between two Gaussian distribution classes. Fig. 12 is the ASC results. It shows that ASC separated the overlapped area. The classification error of ASC is 2.0% and it needs 86 prototypes, i.e. ASC uses 86/10,000 = 0.86% (r_c = 0.86%) prototypes of 1-NN and gets lower classification error for the overlapped classes.