CHAPTER IV

CUSTOMIZATION FOR CLASSIFICATION

In this chapter, we will propose our frameworks in performing customization for classification. We will also propose some algorithms that behave according to each framework and conduct numerical experiments to evaluate their performances.

4.1 Customization for Classification by Transforming Input Space

Here we will present the first approach. The intuition behind this approach is based on an expectation that the space described by the original classifier and the customization data could be very similar to each other. By being similar, we expect that the optimal decision boundary for customization data can be transformed into the decision space resulted from the original classifier via the operation that preserves its topology, namely both of these spaces are homeomorphic to each other.

4.1.1 Generating sequence of transformation for probabilistic classifiers

4.1.1.1 Algorithm

The main objective of this approach is to determine a mapping $\mu : \mathbb{R}^n \rightarrow \mathbb{R}^n$ to be applied to input data, in order to make the data be better adapted to the classifier. The problem in performing task-based customization for classification is due to the nature of classification that yields discrete output. In the task of classification, there may be many classifiers which classify data correctly. Suppose that there are some data which are not classified correctly and we want to change or customize the classifier on them. Therefore, the issue is how much of the change should be made. If the change is as small as possible, which should be the ideal case when we want to preserve the result from the original classifier, then the decision boundary will be depended mostly on those data and will not have any effect on surrounding the data as much as they should be. Because of the difficulty in determining the proper decision boundary, we

decide to perform the task on probabilistic values of classification results, which can be thought of as performing regression and will be easier for the task-based constraint. Here, we present an algorithm for generating the mapping for a probabilistic binary classifier in Algorithm 1.

```
input : an original classifier, customization data
output: transformation sequence
stepsize ← initial_stepsize;
transform_sequence ← ∅;
curdata ← customization data;
cur_total_prob ← ∑ p(curdata,classifier);
while stepsize > stepsize_limit do
    center ← getnext(halton_sequence);
    direction ← estimate_gradient(center,classifier);
    transformation ← (center,direction,stepsize);
    mapdata ← transform(curdata,transformation);
    map_total_prob ← ∑ p(mapdata,classifier);
    if map_total_prob > cur_total_prob then
        curdata ← mapdata;
        cur_total_prob ← map_total_prob;
        transform_sequence ←
        append(transform_sequence,transformation);
    end
    stepsize ← α × stepsize;
end
```

Algorithm 1: Generating transformation sequence.

The mapping is separated into many transformation sequences, for its flexibility in order to achieve complicated nonlinear transformation. As shown in Algorithm 1, it will pick the center of each transformation from Halton sequence, to guarantee coverage of the input space. Then according to the current *stepsize* and estimated gradient, we will have a candidate for transformation described by three factors, i.e. the center of transformation, direction and magnitude. The transformation will be kept if it is likely to lead to better result. The change on space around the center of transformation is calculated as being

influenced by the attempt to move the space at center of transformation to a new position. For simplicity of calculation, we let the change be in the same direction to the change on the center, while the magnitude of the change is calculated by a distribution, proportional to the size of the change on the center. To achieve convergence, *stepsize* is gradually reduced in each loop.

In order to improve the algorithm, we further consider about the problem with imbalance on the number of training data, i.e. there are some classes with overwhelmingly more or less number of training instances. A problem may arise when using the proposed algorithm by summing up the probabilistic values of predictions resulted from all customization data, which could cause the algorithm to be over-biased on some classes with high number of training instances, as shown in Figure 4.1. In order to be better cope with this problem and to lessen the difference from getting probabilistic outputs from different type of classifiers, we introduce the idea of using an output function. The purpose is to assign more importance to the data with higher probabilistic values, which gives the effect that all customization data will be more likely to be correctly classified, and lessens the effect which will mislead the decision boundary by other customization data. Therefore, instead of using probabilistic values directly, in this paper, we propose using the value from output function $s(x) = 1 - ((1 - p(x))^2)$ which its shape is as shown in Figure 4.2.

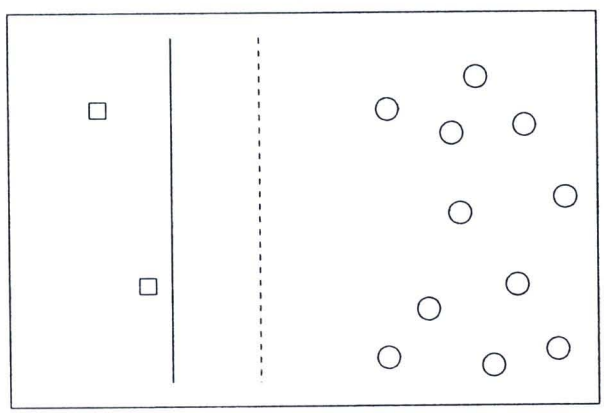


Figure 4.1: Visualization of the problem in determining objective for mapping. The class of data are shown by depicting each of them as square and circle. The solid line is a likely decision boundary resulted from summing the objective value calculated from each data and the dashed line is the most suitable decision boundary.

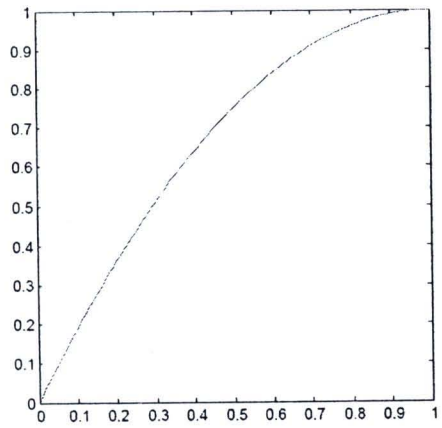


Figure 4.2: Plotting of output function $s(x) = 1 - ((1 - p(x))^2)$.

4.1.1.2 Numerical Results

To demonstrate the usefulness of this algorithm, we generated experimental data in two dimensions. The result for the original probabilistic classifier was determined by equation $p_1 = \frac{x+y}{\sqrt{2}}$. The decision boundary was a straight diagonal line separating half of the input space, given that the input data was normalized into the range of $[0, 1]^2$. The probabilistic value was directly proportional to the distance measured from the decision boundary, with the minimum and maximum values of 0 and 1, respectively. On the other hand, customization data was generated from different distribution, determined by whether values of both attributes were less than 0.7. We applied Algorithm 1 with the parameter *initial_stepsize*= 0.1, *stepsize_limit*= 0.001, $\alpha = 0.99$ and with changing impact as exponential function: $e^{-x/\beta}$ where $\beta = 1$. We conducted an experiment using the mentioned probabilistic classifier, and a weighted nearest neighbor classifier generated from randomly sampled data according to the decision boundary from the equation, using $1/x^k$ as weight function where k is the number of attributes, considering only 10 nearest neighbors for each class. These setting of parameters will be used as default value in the rest of this work. The result is shown in Table 4.1 and can be visualized as in Figure 4.3.

In Table 4.1, the result from classifying mapped customization data using the original classifier yielded some improvement compared to the result from classifying customization data. This means that the process transforms

Table 4.1: Accuracy comparison between customization data, mapped customization data and mapped customization data using the proposed output function.

	CUST DATA	MAPPED CUST DATA	OUTPUT FUNCTION
ORIGINAL CLASSIFIER	82.40	82.90	**87.70**
WNN	87.00	85.30	**89.00**

the space of customization data to be better fit with the space of the original classifier. However, when we tried using data sampled from the original classifier to create a probabilistic classifier, which was weighted nearest neighbor (WNN), the result from using the weighted nearest neighbor for mapped customization data became worse due to misleading of the probabilistic value from voting. By using the concept of output function, the problem was lessened and the result was shown with obvious improvement.

4.1.2 Applying with other classifiers

4.1.2.1 Algorithm

In order to apply the algorithm for probabilistic classifiers to all non-probabilistic classifiers, we propose an algorithm to create a probabilistic classifier from the former one, for the sole use of generation of transformation sequence. To treat the original classifier as a black box, we have to perform sampling and classify them using the classifier, whose result will be used as data to train the generated probabilistic classifier. Moreover to keep the influence from the nature of the classifier model at minimum, we decide to use an instance-based classifier. This results in the following algorithm to create a weighted nearest neighbor through sampling for a binary classifier in Algorithm 2.

input : original classifier

output: weighted nearest neighbor classifier

```
1  NN_data ← ∅;
2  NN_weight ← ∅;
3  while size(NN_data) < datasize do
4  |  newdata ← getnext(halton_sequence);
5  |  predictweight ← classify(newdata,NN_data,NN_weight);
6  |  if predictweight <0.5 then
7  |  |  NN_data ← append(NN_data,curdata);
8  |  |  NN_weight ← append(NN_weight,1-(2 × predictweight));
9  |  end
10 end
```

Algorithm 2: Generating a weighted nearest neighbor classifier from an existing classifier.

The main idea for this algorithm is that the newly generated data will be classified by the current classifier. If the classification result is wrong, then the current classifier will be improved by adding the newly generated data as a new instance. The weight of the instance is determined by the weight from the classification result.

For the generated classifier to be appropriately useful, there are two conditions which should be satisfied. Firstly, the classifier should be able to provide probabilistic prediction for each class, in a meaningful sense. Secondly, its decision boundary should be similar to that of the original classifier at least up to some degree. The sampling approach used in the algorithm implies curse of dimensionality, i.e. the amount of sampling required to satisfy the second condition exponentially increases with the number of attributes. In order to fix this problem, we decide to use the method of combining the result from the generated classifier with the original one.

Given that the weight for the original classifier is higher, the combined classifier will obviously predict the same result as the original, and the generated classifier will act as an alteration on probabilistic values, as in the following equation.

Table 4.2: Accuracy from the generated classifier when using the algorithm that is applicable to non-probabilistic classifier.

CUSTOMIZATION DATA	82.40
MAPPED CUSTOMIZATION DATA	82.40
MAPPED WITH OUTPUT FUNCTION	83.20

$$p_{new} = ((1+\alpha)p_{original} + p_{generated})/(2+\alpha).$$

Where α is a small positive real. The drawback of this method is that in order to achieve the same classification result as the original classifier, we have to trade that off with the continuity on the probabilistic value from the generated classifier.

We performed an experiment to evaluate the generated classifier in the previous subsection, using Algorithm 2 with datasize equal to 2,000. We replaced the original classifier with the generated one while all other parameters were set in the same way as in the previous experiment. The result is shown in Table 4.2.

Comparing results between Table 4.1 and Table 4.2, it is not surprising that the result from using the algorithm that is applicable to non-probabilistic classifier was inferior due to the additional error resulted from having the additional step in generating a probabilistic classifier from non-probabilistic one. However, the result in Table 4.2 still show slight improvement compared to the result before mapping.

4.1.2.2 Numerical Results

We conducted an experiment to demonstrate the improvement made by the algorithm. In order to measure the performance we used the real-world datasets obtained from the UCI machine learning repository (Asuncion and Newman, 2007), *pendigits*, representing the tasks of classifying numeral digits for handwriting recognition, to demonstrate the task in adapting the classifier to another similar dataset. In the reality, the input data from handwriting recog-

Table 4.3: Accuracy of each transformed subclassifier.

	ORIGINAL	TRANSFORMED
WNN	99.39 ± 1.03	$\mathbf{99.69 \pm 0.57}$
NEURAL NET	98.86 ± 1.42	$\mathbf{99.40 \pm 0.98}$
SVMS	99.46 ± 0.68	$\mathbf{99.59 \pm 0.66}$

nition is sequence of pen strokes, which is sequence of time series. However, all 16 attributes in the *pendigits* dataset were extracted features from this sequence of time series and we used these values for classification. The original training data was used to train the classifier, while the original test data, being generated from different group of users, was separated into two groups for the task of customization and testing. The original training data and the other set of data had 7,494 and 3,498 instances respectively which were almost equally distributed to 10 classes representing digits 0-9, and for each class, we assigned 50 instances as customization data. As a result, we had roughly 750 training data, 50 customization data and 300 test data for each class. The setting of parameters was the default value as stated in the previous subsection and all of data were normalized to be in the range of $[0, 1]$.

In this experiment, we used three types of classifiers: weighted nearest neighbor, neural networks (Mitchell, 1997; Hastie et al., 2001; Michie et al., 1994; Han and Kamber, 2000) and support vector machines (Cristianini and Shawe-Taylor, 2000), to create one-against-one classifiers between each pair of classes to which the customization was applied. Then these subclassifiers were combined by two methods of voting and decision trees. So, there were 45 subclassifiers for each type which were trained with roughly 1,500 training data. We performed customization on each subclassifier with 100 customization data and the experimental result was taken from about 300 test data for the experiment on subclassifiers and all 2,998 test data for the combined classifier. The results are as shown in Table 4.3 and Table 4.4.

An important note is that in Table 4.3, the value of standard deviation was not from cross validation but was the standard variation from the classifier of each class. Thus, high value of standard deviation means that there were high

Table 4.4: Accuracy of the combined classifier created from transformed subclassifiers.

COMBINING METHOD	ALGORITHM	ORIGINAL	TRANSFORMED
MAX WINS	WNN	97.80	**98.13**
	NEURAL NET	95.43	**96.90**
	SVMS	97.67	**98.37**
ADAG	WNN	97.80	**98.10**
	NEURAL NET	95.76	**97.73**
	SVMS	97.67	**98.37**

Table 4.5: Accuracy of each transformed subclassifier when using generated subclassifiers to determine the transformation.

	ORIGINAL	TRANSFORMED
NEURAL NETWORK	**98.86 ± 1.42**	98.72 ± 1.39
SVMS	**99.46 ± 0.68**	98.71 ± 1.61

differences in difficulties in classifying each pair of digits. This fact is obvious since some digits like 1 and 7 are harder to classify apart with writing strokes, compared to classification between 0 and 4 which there are differences in the numbers of pen strokes, curve of the pen strokes and direction of pen strokes. For the experiment on the method that is appliable to non-probabilistic classifiers, we used Algorithm 2 to generate weighted nearest neighbor classifiers with 2,000 data. The results are shown in Table 4.5 and Table 4.6.

In Table 4.5 and Table 4.6, we did not conduct the experiment with weighted nearest neighbor on this matter since it is more realistic to use the data used in weighted nearest neighbor itself instead of generating new data. The results obviously imply that the generated probabilistic classifier obtained by transformation is worse than the original classifier. However, there still exists the problem about curse of dimensionality as previously stated. Since *pendigits* has sixteen attributes then it will require at least $2^{16} = 65,536$ for the data to exist in every orthants; each sector determines the positiveness of the value in each axis. So it can be seen by how sparse the generated dataset with 2,000 instances is in the sixteen dimensional space.

Table 4.6: Accuracy of the combined classifier created from transformed subclassifiers when using generated subclassifiers to determine the transformation.

COMBINING METHOD	ALGORITHM	ORIGINAL	TRANSFORMED
MAX WINS	NEURAL NETWORK	95.43	95.00
	SVMS	97.67	94.96
ADAG	NEURAL NETWORK	95.76	94.80
	SVMS	97.67	94.70

Though the result was not good for data with many attributes, in the case of a small number of attributes, this method was still able to yield satisfying result. The other issue is that though the classifiers are already probabilistic but each probabilistic classifier may return the value of probability in different sense so it may be a good idea to also use this method to generate a probabilistic classifier to guarantee the worst case in determining transformation sequence.

4.1.3 Overall Framework

Combining with the algorithm previously mentioned, we will get the framework of customization for classification as shown in Figure 4.4. All diagrams in our works follow this description. A block with rectangular shape represents a process and an oval shaped block represents data. A solid arrow represents the relation of being input to a process while a dashed arrow represents the relation of a process generating its output. A dashed box covers parts of the diagram that are kept as important result for continual usage. In this diagram, the box also shows the process within a customized classifier.

The framework starts with the original classifier which we want to perform customization on. Next, we generate a probabilistic classifier to be used in the process from the customization data and the original classifier. Then we determine the transformation sequence to make the customization data be better fit to the generated classifiers (since the generated classifiers have the same decision boundary to the original classifiers, the transformed customization data will also be better fit to the original classifiers as well). The customized classifier consists of the transformation sequence and the original classifier. Each time

classification is performed, the input data will be mapped with the transformation sequence then the mapped data will be classified by the original classifier, giving the prediction.

4.2 Customization for Classification by Patching

Here, we will present another alternative approach. The goal of this approach is to use customization data to classify if the result from the original classifier is trustable in a region. Otherwise, we will classify it with the other classifier created from the customization data instead. In our case, we will combine these processes into a classifier called patcher, which contains the same set of classes as those of the original classifier with one more class representing that the answer should be left to the original classifier. The reason that we name this approach patching is because this method provides the intuition that the result from the original classifier will be corrected in untrustable region by patching it with the result from another classifier, while the result in trustable region which needs no correction will be left as is. Thus, this gives the intuition as the original classifier is patched to fix the bad result.

4.2.1 Algorithms

We will present four algorithms which belong to this approach in this subsection.

input : original classifier, customization data
output: patching classifier
1 NN_data ← customization_data;
2 **foreach** *data* in NN_data **do**
3 | **if** classify_1nn(*data*,NN_data) = *data.class* **then**
4 | | *data.class* ← original_classifier;
5 | **end**
6 **end**

Algorithm 3: Patching using 1-nearest neighbor.

Algorithm 3 is the simplest among the four algorithms of this group which will be presented within this section. The idea is to use 1-nearest neighbor and

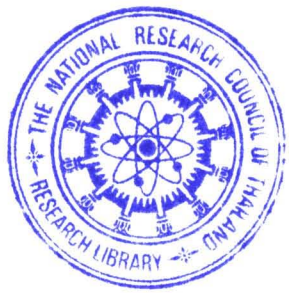
use the result from the original classifier if the data from 1-nearest neighbor is classified by the original classifier correctly.

```
input : original classifier, customization data
output: patching classifier
1  data_queue ← customization_data;
2  NN_data ← ∅;
3  correctset ← ∅;
4  while notempty(data_queue) do
5      data ← dequeue(data_queue);
6      if classify_1nnpatcher(data,NN_data,original_classifier)
       = data.class then
7          correctset ← append(correctset,data);
8      end
9      else
10         if classify(data,original_classifier) = data.class then
11             data.class = original_classifier;
12         end
13         NN_data ← append(NN_data,data);
14         foreach correctdata in correctset do
15             if
               classify_1nnpatcher(correctdata,NN_data,original_classifier)
               ≠ data.class then
16                 remove(correctset,correctdata);
17                 enqueue(data_queue,correctdata);
18             end
19         end
20     end
21 end
```

Algorithm 4: Patching using 1-nearest neighbor with the reduced number of instances.

Algorithm 4 is a slight modification version of Algorithm 3 in order to reduce the number of data used in the 1-nearest neighbor patcher. The idea is to add one data into the patching classifier at a time, and try not to add it if it

is classified correctly. Different from Algorithm 3, the classifier resulted from using Algorithm 4 will also depend on the order of data used in the algorithm. The procedure which we try to reduce the number of training data for 1-nearest neighbor will reduce the time used in classification and will also reduce the complexity of decision boundary, which might yield better result as well.



input : original classifier, customization data
output: patching classifier

```
1  data_queue ← customization_data;
2  patch_data ← ∅,correctset ← ∅;
3  while notempty(data_queue) do
4      data ← dequeue(data_queue);
5      if classify_patcher(data,patch_data,ori_classifier) = data.class then
6          correctset ← append(correctset,data);
7      else
8          if classify_patch(data,patch_data) = ori_classifier then
9              data.radius ← find_radius(data,patch_data,correctset);
10             patch_data ← append(patch_data,data);
11         else if classify(data,ori_classifier) ≠ data.class then
12             data.radius ← find_radius(data,patch_data,correctset);
13             foreach ball in patch_data do
14                 ball.radius ← reduce_radius(ball, data);
15             end
16             patch_data ← append(patch_data,data);
17         else
18             foreach ball in patch_data do
19                 ball.radius ← reduce_radius(ball, data);
20             end
21         end
22         foreach correctdata in correctset do
23             if classify(correctdata,ori_classifier) ≠ correctdata.class then
24                 remove(correctset,correctdata);
25                 enqueue(data_queue,correctdata);
26             end
27         end
28     end
29 end
```

Algorithm 5: Patching with a reduced number of hyperspheres.

Algorithm 5 will create open balls or hyperspheres without boundary

with customization data as their centers. The result within the region of each ball is classified as belonging to the same class as the customization data which is the center of the ball. The result for undefined region is taken from the original classifier. The algorithm will perform similarly to Algorithm 4 except that the region of the hypersphere requires different handling. First of all, each hypersphere must not intersect with others, and that is why when there are some changes in patch_data, the algorithm will require more complicated procedure. Note that there are two functions that require more description in Algorithm 5. For the first one, *reduce_radius* will shrink down an existing hypersphere if the hypersphere is the cause of wrong classification or if the two arguments intersect with each other. If the wrong classification is caused by a new hypersphere then it will make the two hyperspheres become almost touched, and if it is a new data in different class then it will make the radius of the hypersphere half of the distance between the two instances. For the second one, *find_radius* will be used when a new hypersphere or data is added. The function will try to make the new hypersphere as large as possible with some constraints that it must not intersect with other hyperspheres and that its radius is not more than half of the distance between its center and other instances not in any patch region. Since this may not be very obvious we will provide a proof that this algorithm will terminate as in Proposition 1.

Proposition 1. *Algorithm 5 will eventually terminate.*

Proof. In each loop, the algorithm will dequeue and perform at least one of the following actions; add data from the queue into *correctset* in the case that the data is correctly classified by the current classifier or add data from the queue to *patch_data* then reduce radius of an existing hypersphere in order to fix the result of the current classifier. Each time enqueue is performed, the data must be taken from *correctset*, which causes the total numbers of data in the queue, *correctset* and *patch_data* to always be the same. In each loop when enqueue is performed, it must also add another instance to *patch_data* or reduce the size of a hypersphere. Adding an instance to *patch_data* will cause the total number of instances in *correctset* and the queue to be lower as well and will eventually make the queue empty. The size of the hypersphere can be reduced for a limited number of times, bounded by the total number of instances. And if the radius

is at the lowest possible, then there cannot be an intersection with any other data, which causes the loop to perform other action instead. When both of these actions are not performed any longer then the number of instances in the queue cannot be increased. Hence, the loop will terminate. ■

input : original classifier, customization data
output: patching classifier
1 patch_data ← customization_data;
2 **foreach** *data* in patch_data **do**
3 | $s \leftarrow$ patch_data which $(.class \neq data.class)$;
4 | $data.radius \leftarrow min_{x \in s} d(data, x)/2$;
5 **end**

Algorithm 6: Patching with intersectable hypersphere.

Algorithm 6 is different from the rest in that it allows intersection of regions, as long as they are both of the same class. The algorithm will make the radius of each hypersphere half of the distance between the center of the hypersphere and the closest instance in a different class.

4.2.2 Numerical Results

In this section, we conducted experiments on the *pendigits* dataset and the classifiers in the previous section. Also, to better demonstrate the result, we decided to also use other datasets from the UCI machine learning repository called *optdigits*, representing the task of optical character recognition. The dataset *optdigits* represents images of digits 0-9 in 8x8 pixels, resulted in 64 attributes each of which is an integer in the range of 0-16 reflecting grayscale value of the images. The dataset was separated by the group of writers in the same way as *pendigits*, consisting of 3,823 instances of original training data and 1,797 instances that were used for customization and testing. For each class, 50 instances were assigned as customization data, and thus we had roughly 380 training data, 50 customization data and 130 test data for each class. We used principal component analysis to reduce the number of attributes to 16, the same as in *pendigits*, and then normalized the value of these 16 attributes to be in the range $[0, 1]$ before preparing the data for the experiments in the same way. The

Table 4.7: Accuracy of patched subclassifiers from *pendigits* data.

	ORIGINAL	ALGO 3	ALGO 4	ALGO 5	ALGO 6
WNN	99.39 ± 1.03	99.62 ± 0.83	99.62 ± 0.82	99.61 ± 0.85	**99.72 ± 0.74**
NN	98.86 ± 1.42	99.41 ± 0.74	99.46 ± 0.71	99.26 ± 0.93	**99.51 ± 0.84**
SVMs	99.46 ± 0.68	99.63 ± 0.45	99.65 ± 0.47	99.63 ± 0.43	**99.75 ± 0.35**

Table 4.8: Accuracy of patched subclassifiers from *optdigits* data.

	ORIGINAL	ALGO 3	ALGO 4	ALGO 5	ALGO 6
WNN	99.58 ± 0.90	99.53 ± 0.92	99.62 ± 0.82	99.61 ± 0.85	**99.72 ± 0.74**
NN	98.22 ± 2.02	98.23 ± 2.03	99.46 ± 0.71	99.26 ± 0.93	**99.51 ± 0.84**
SVMs	**99.80 ± 0.37**	99.78 ± 0.39	99.65 ± 0.47	99.63 ± 0.43	99.75 ± 0.35

experiments on combined classifiers are shown in Table 4.7, Table 4.8 and Table 4.9. Since the algorithm can be applied to multiclass classifiers without any modification, we also conducted an experiment on combined classifiers, and the result is shown in Table 4.10.

The result from Table 4.7, Table 4.8, Table 4.9 and Table 4.10, show that the proposed algorithm improved the classification result in most cases, Algorithm 6 provided the best result among the four proposed algorithms which perform customization for classification by patching. An interesting issue is the result from applying the algorithms with multiclass classifiers. In Table 4.9 and Table 4.10, applying the algorithms with multiclass classifiers brought worse result in most cases, with the exception of Algorithm 3. For Algorithm 4 and Algorithm 5, their results depended on the order of the data used in the algorithms, and applying the algorithms with multiclass classifiers gave more bias resulted from the order of the data. For Algorithm 6, as the radius of the ball was calculated from the distance to the other data with different class, using the algorithm with multiclass classifiers provided smaller radius of the hyperspheres compared to using the algorithm on each subclassifier because the number of data in different classes had increased.

Now we compare both frameworks of transforming input space and patching. Transforming input space seems like a good approach in performing con-

Table 4.9: Accuracy of combined classifier from patched subclassifier.

	ALGORITHM	ORIGINAL	ALGO 3	ALGO 4	ALGO 5	ALGO 6
PENDIGITS MAX WINS	WNN	97.80	98.10	98.10	97.97	**98.50**
	NEURAL NET	95.43	96.83	97.36	96.46	**97.67**
	SVMS	97.67	98.37	98.50	98.17	**98.63**
PENDIGITS ADAG	WNN	97.80	98.20	98.10	98.03	**98.50**
	NEURAL NET	95.76	97.30	97.40	97.03	**97.73**
	SVMS	97.67	98.33	98.07	98.17	**98.60**
OPTDIGITS MAX WINS	WNN	97.22	97.07	96.38	97.22	**97.30**
	NEURAL NET	93.60	93.99	94.06	93.83	**94.37**
	SVMS	98.54	98.54	98.15	**98.61**	98.54
OPTDIGITS ADAG	WNN	97.22	97.07	96.68	97.22	**97.30**
	NEURAL NET	92.60	93.06	92.37	92.75	**93.29**
	SVMS	98.54	98.54	98.38	**98.61**	98.54

tinuous changes to create a more suitable classifier. However, an obvious drawback from transforming input space is the high number of parameters of the algorithm which is also due to applying the algorithm to the task of classification which yields discrete result. In contrast, patching is much simpler and has few parameters which cause it to have much less problem in overfitting. The idea of patching is to use the result from each of two classifiers, the original one and the one created from customization data. We can give some estimation like lower bound or expected performance of a classifier resulted from patching with both classifiers. Suppose that we determine whether we should believe the original classifier by random then the expected accuracy of the new classifier should be the average value of the accuracy from both classifiers. Moreover, by the fact that our decision is based on a plausible reasoning, the resulted classifier from patching should obviously yield better result than the average accuracy of both classifiers. Comparing the result from Table 4.4 and Table 4.9 which are the results from transforming input space using probabilistic value of *pendigits* and patching, patching by Algorithm 6 yields better result. However, this is due to difficulty in transforming input space. In summary, transforming input space is an approach with high potential but is hard to achieve and patching is a simple method but has limited potential in both upper bound and lower bound of the result.

Table 4.10: Accuracy of patched combined classifier.

	ALGORITHM	ORIGINAL	ALGO 3	ALGO 4	ALGO 5	ALGO 6
PENDIGITS MAX WINS	WNN	97.80	**98.40**	97.97	98.17	98.27
	NEURAL NET	95.43	**97.73**	97.40	96.83	96.83
	SVMS	97.67	98.27	97.30	98.10	**98.30**
PENDIGITS ADAG	WNN	97.80	**98.40**	97.97	98.17	98.27
	NEURAL NET	95.76	**98.03**	97.26	97.20	97.26
	SVMS	97.67	98.27	97.30	98.10	**98.30**
OPTDIGITS MAX WINS	WNN	97.22	97.07	95.07	97.22	**97.30**
	NEURAL NET	93.60	**93.99**	89.67	93.83	**93.99**
	SVMS	98.54	98.46	96.92	98.61	**98.69**
OPTDIGITS ADAG	WNN	97.22	97.07	95.07	97.22	**97.30**
	NEURAL NET	92.60	**93.06**	88.97	92.91	**93.06**
	SVMS	98.54	98.46	96.92	98.61	**98.69**

4.3 Customization for Classification with Nonlinear Dimensionality Reduction

This framework is based on the idea of transforming input space as well, but with a different method in finding the transformation. Instead of finding sequence of small transformations, this framework will use dimensionality reduction to find the transformation between both input spaces. This framework is shown in Figure 4.5 and the resulted classifier is shown in Figure 4.6.

The framework begins with generating data that represents the original classifier by randomization for its attribute values and uses the class from prediction of the original classifier. After that, we will embed the space of this data and the customization data into a higher-dimensional space. We then use supervised linear dimensionality reduction algorithms, such as KMMLN, with the expectation that it will align both data sets on each other without changing their topology. After that we will perform regression such that the data in that space should be mapped back into the space of the original classifier based on the value of the generated data. The new classifier will work by transforming the test data into the space that aligns it with the data belonging to the original classifier. After that, the values of attributes of the data in the original space corresponding to the test data will be calculated by regression, and that attribute values will be used by the original classifier for prediction.

4.3.1 Numerical Results

We show the experimental result of the proposed framework. For visualization, we used 2-dimensional data with no noise and with two possible classes, to test how the framework performed on each kind of basic linear transformation. We used the same customization data and test data while varying the original classifier which is 1-nearest neighbor generated from the space that can be mapped onto customization data perfectly by translation, scaling, and rotation. All the data in this experiment is shown in Figure 4.7.

The reason for us to choose 1-nearest neighbor as the original classifier is for easiness in the step of generating data that represents the original classifier. In this case, we used the data in 1-nearest neighbor directly. The next issue was how to embed both data into the same space. An easy implementation may be to use translation, or any kind of rigid transformation that makes both datasets not overlap with each other, and even easier, we may add another dimension whose value represents the dataset where each data comes from. The visualization of this is as in Figure 4.8.

Note that, this framework relies mainly on the expectation that the algorithm for nonlinear dimensionality reduction, which is the algorithm in the group of KMMLN in this case, will be able to align both datasets onto each other. To achieve this result, we considered both the method of embedding and the inner process of the KMMLN algorithm. In this experiment, we used the algorithm KDNE, KLMNN and KNCA which are the kernelized versions of linear dimensionality reduction algorithms DNE (Zhang et al., 2007), LMNN (Weinberger and Saul, 2009) and NCA (Goldberger et al., 2005) respectively, and the algorithms for them are already stated in the original paper of KMMLN (Chatpatanasiri et al., 2010, 2008). The results are shown in Table 4.11 and Table 4.12.

The results show the accuracy when using an original 1-nearest neighbor classifier and the accuracy of 1-nearest neighbor with customization data. There is clearly significant improvement compared to the original classifier, and some improvement compared to 1-nearest neighbor on customization data. How-

Table 4.11: Result from customization using linear dimensionality reduction algorithm.

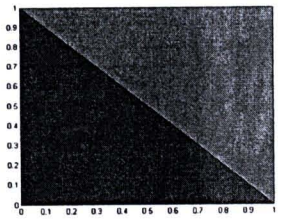
	ORIGINAL	CUSTOM DATA	DNE	LMNN	NCA
TRANSLATION	83.6	**93.4**	83.6	83.5	83.7
SCALING	86.0	**93.4**	86.0	86.1	86.3
ROTATION	88.6	**93.4**	88.6	88.6	88.6

Table 4.12: Result from customization using the KMMLN algorithm.

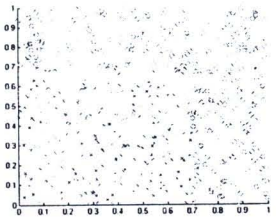
	ORIGINAL	CUSTOM DATA	KDNE	KLMNN	KNCA
TRANSLATION	83.6	93.4	87.0	93.8	**97.9**
SCALING	86.0	93.4	87.7	**94.1**	87.9
ROTATION	88.6	93.4	91.0	**93.6**	91.1

ever, two issues need to be noted. First is that all the three datasets in this experiments are simple linear transformation from a dataset with really simple decision boundary, as shown in Figure 4.8. The second issue is that though the numerical result looks fine but from visualization, the two datasets did not align on each other when using KMMLN, with the exception of performing KNCA on the translation dataset which yielded much more significant difference compared to other cases in the experiment. Thus, the algorithm may not have much potential in higher dimensional datasets with more complicated decision boundary.

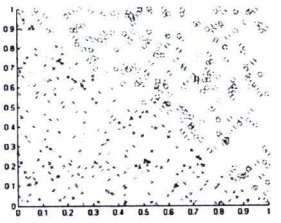
In conclusion, the framework seems to have the potential to yield satisfying result, but we could not bring experimental results as we expected yet. So we only present this framework here as it may be a good idea for further development in this line of work.



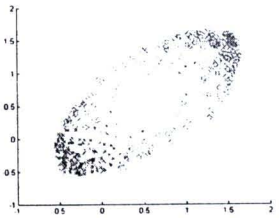
(a) Original classifier



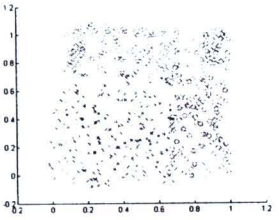
(b) Customization data



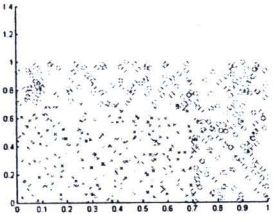
(c) Randomly generated data from the original classifier to be used for weighted nearest neighbor



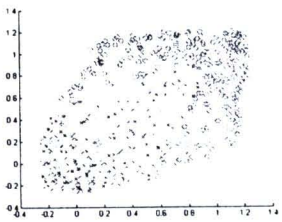
(d) Mapped customization data, using the original classifier



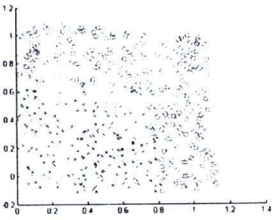
(e) Mapped customization data, using the randomly generated weighted nearest neighbor



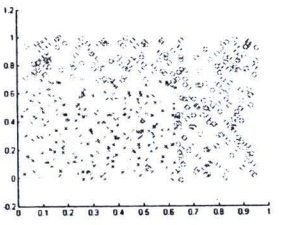
(f) Mapped customization data, using the new generated classifier from the proposed algorithm



(g) Mapped customization data, using the original classifier, with the output function



(h) Mapped customization data, using the randomly generated weighted nearest neighbor, with the output function



(i) Mapped customization data, using the new generated classifier from the proposed algorithm, with the output function

Figure 4.3: Visualization of results from transforming generated probabilistic classifiers.

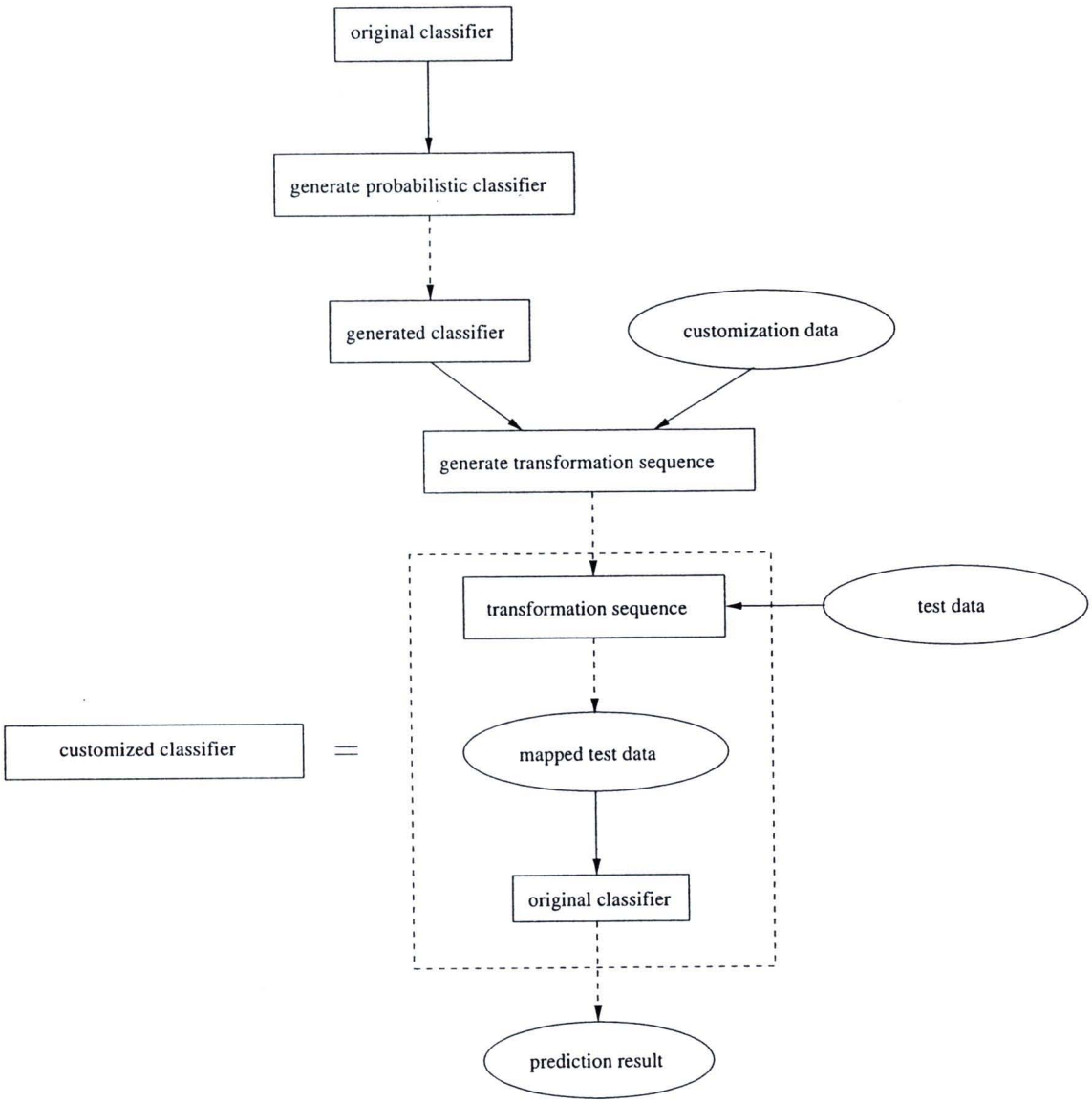


Figure 4.4: Framework of customization for classification by Transforming Input Space.

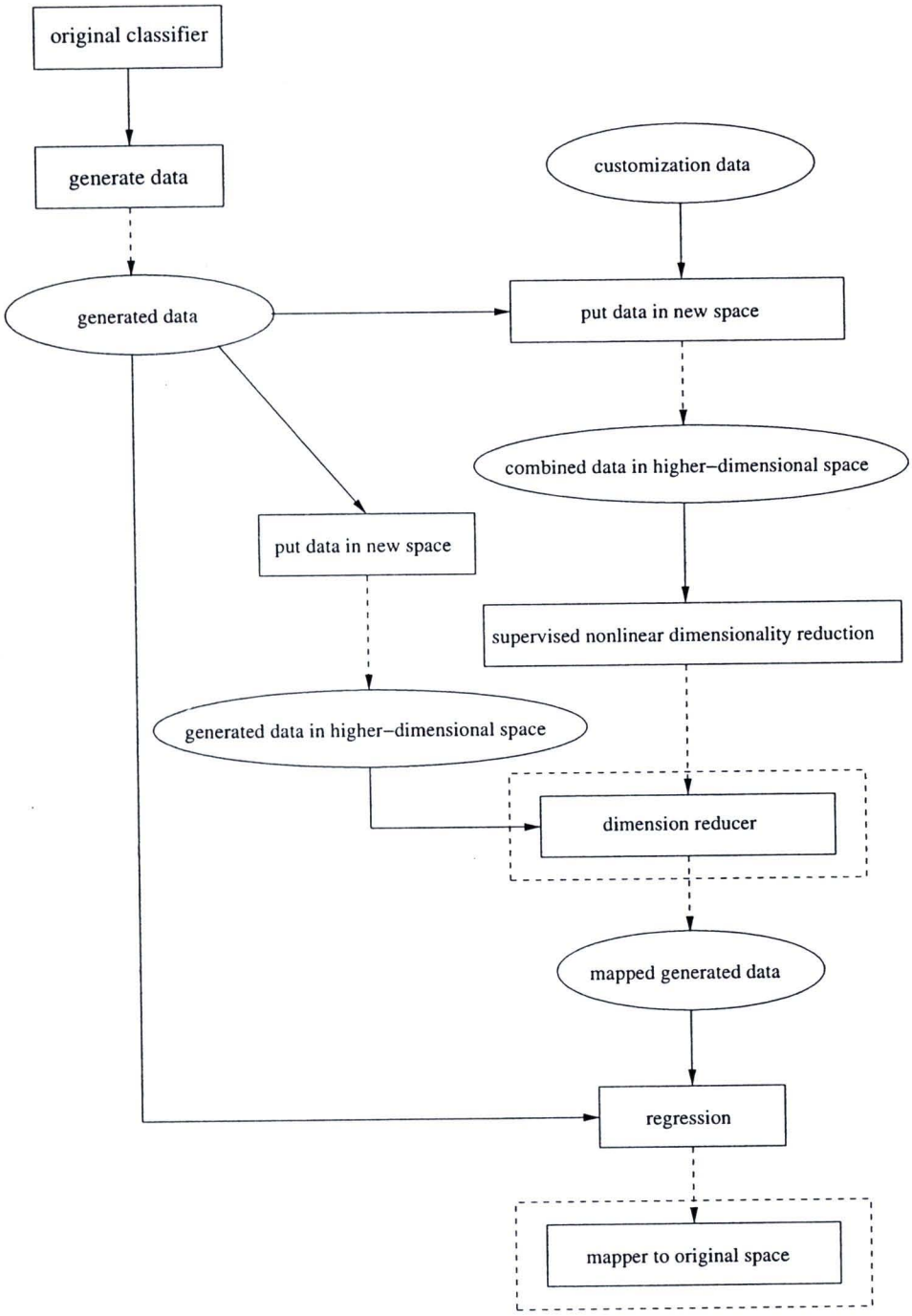


Figure 4.5: Framework of customization for classification with nonlinear dimensionality reduction.

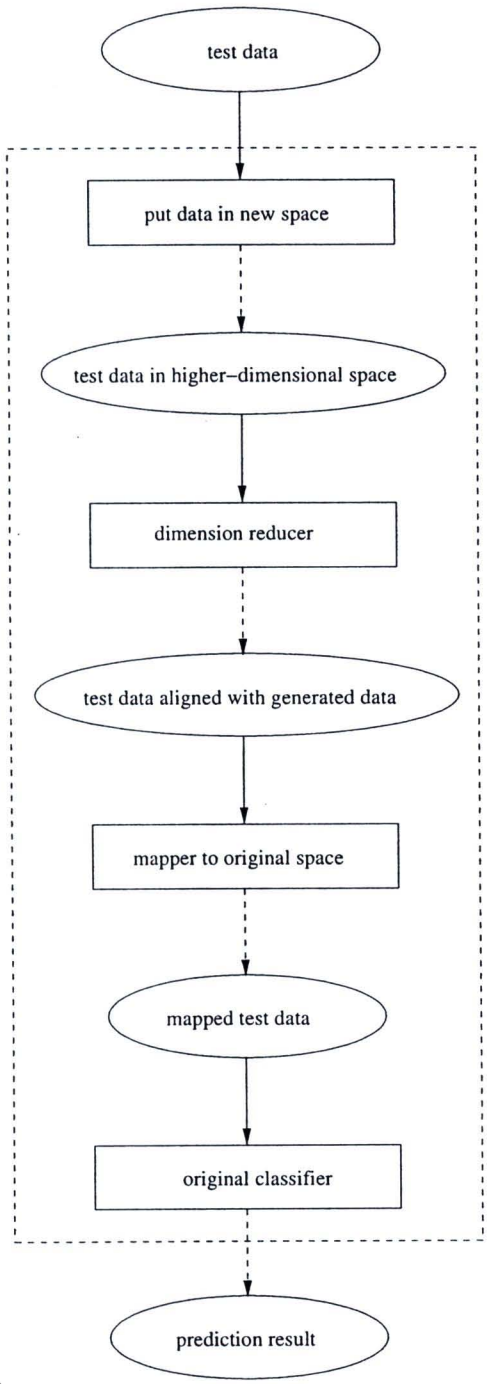
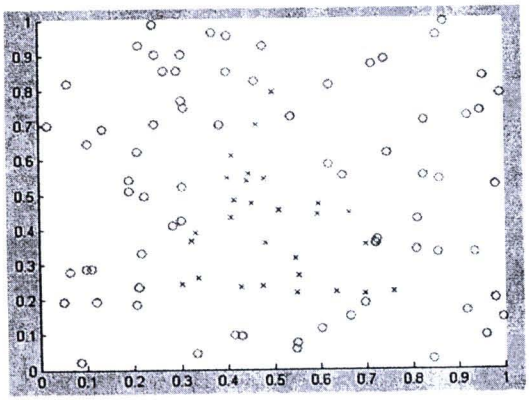
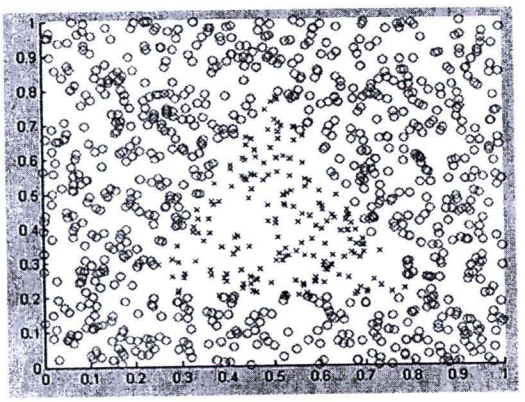


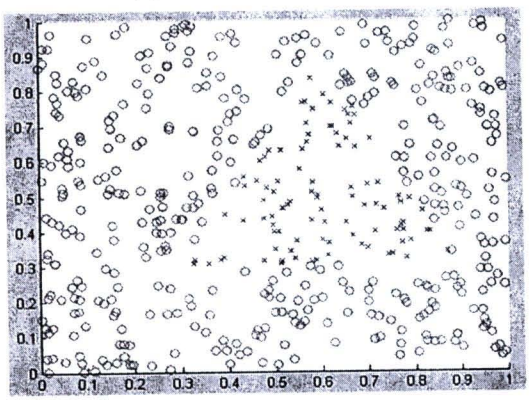
Figure 4.6: New classifier from the framework for customization of classification with nonlinear dimensionality reduction.



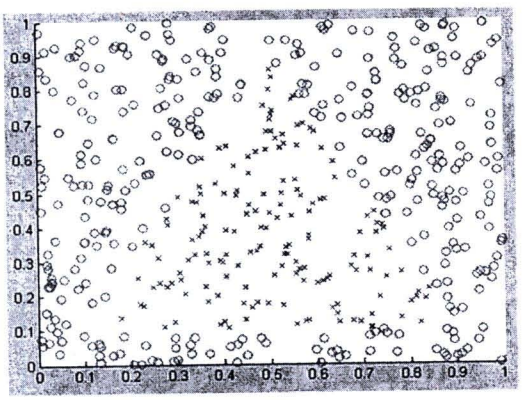
(a) Customization data



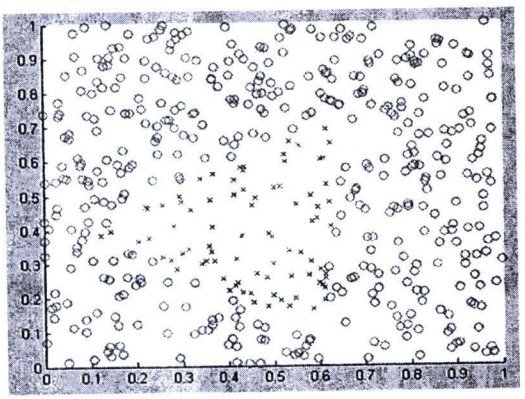
(b) Test data



(c) Original data for 1-nn by translation



(d) Original data for 1-nn by scaling



(e) Original data for 1-nn by rotation

Figure 4.7: Visualization of generated dataset.

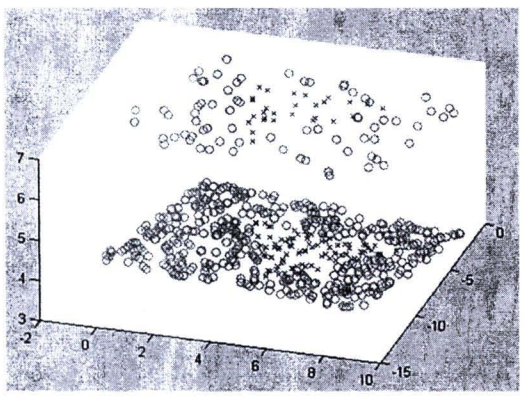


Figure 4.8: Embedding the data into a higher dimension.