1997

Optical character categorization: Clustering as it applies to OCR

Jennifer Greenwald

Rochester Institute of Technology
Computer Science Department

Optical Character Categorization: Clustering as it Applies to OCR

by

Jennifer B. Greenwald

A thesis, submitted to the Faculty of the Computer Science Department in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Approved by:

Dr. Peter G. Anderson

Dr. Fereydoun Kazemian

Dr. Stanislaw P. Radziszowski

7 November, 1997

Optical Character Categorization: Clustering as it Applies to OCR

I, Jennifer Greenwald, prefer to be contacted each time a request for reproduction is made. If permission is granted, any reproduction will not be for commercial use or profit. I can be reached at the following address:

193 Golden Rod Ln.
Rochester, NY 14623

24 February, 1998

1.0 Abstract

I applied clustering analysis to the problem of creating tagged training data for optical character recognition (OCR). The creation of labeled character data by hand is a slow and cumbersome process. My belief is that clustering methods can be applied to character data before tagging it, allowing the operator to label entire groups of characters at once and greatly speeding the time in which tagged character data can be generated. This thesis will provide proof of concept as a basis for more in depth research and eventually the creation of a sophisticated application utilizing these techniques for the generation of labeled training data for OCR systems.

Keywords: Clustering, OCR, Genetic Algorithms, Kohonen SOM, K-Means

Computing Review Subject Codes: Applications [Pattern Recognition]

Table of Contents

2.0 Introduction

My area of interest is clustering, specifically using Genetic Algorithms (GAs), the K-means clustering algorithm, and neural networks as applied to the problem of optical character categorization.

The problem I have investigated is this: in the area of OCR, systems are in place which are capable of doing an adequate job of recognizing text they have been trained for. However, generating training data for these systems (often based on neural networks) is time consuming and expensive, as it currently requires a human operator to sort through thousands of character samples and label each one with the character it represents. This paper reports on some preliminary proof of concept work for a method which could help drastically reduce the amount of human effort involved in tagging this training data.

The idea is to pre-sort the sample characters to be tagged. Once the characters to be tagged are grouped into sets of like characters, the human operator can label them all at once, doing only a few incorrectly classified samples by hand. In a best case scenario, sets of characters which have been clustered together can be put on the screen for an operator all at once, sorted in order of percentage correlation.

I investigated using the K-means clustering algorithm, a genetic algorithm and a type of neural network known as a Kohonen Self-Organizing Map (SOM) to cluster handwritten character data. Each of these methods has shown success in clustering other simpler data. This paper reports the results of using these techniques on handwritten digits (0-9), comparing the success of each method and indicating which is the quickest and the most accurate.

I have not seen any previous work on this topic specifically, though a great deal of research has been done on which features are most useful to represent character data. That work is somewhat applicable here, as the choice of features is likely to have an impact on the success of the method. My literature search was conducted on the WWW and using the RIT library's search tool Einstein, focusing on the keywords OCR, genetic algorithms, Kohonen maps, and document processing.

3.0 Theory

3.1 K-means algorithm

K-means, also knows as C-means or ISODATA, is an iterative clustering algorithm. In order to cluster a set of feature vectors into 'n' clusters, a set of initial cluster centers is chosen. They are often selected randomly from the data set. Once the centers are selected, the distance between each cluster center and each other feature vector is calculated according to some distance metric, and each vector is assigned to the cluster whose center it is closest to. For each cluster, the mean value of the vectors assigned to it is calculated. If these mean values are the same as or very close to the original cluster center, the feature vectors have been correctly

classified and the process has converged. Otherwise, the algorithm replaces the old cluster centers with the new mean values, returns to the distance calculation step, and carries on from there. The success of this process is reliant on the distance metric used, the features chosen, and the initial cluster centers chosen.

3.2 Genetic Algorithms

Genetic algorithms are a product of biologically based computing. A good solution to a problem is found by randomly generating a large population of possible solutions (generally encoded in a string of numbers, often a binary string), evaluating each of them by a problem-specific fitness metric, and then 'mating' the better individuals to make children. This process repeats through a large number of iterations and, if all goes well, produces an individual at the end with a very good fitness rating which is a correct solution to the problem. As so much of the process is based on random numbers, that randomness can often be the life or death of the process. A genetic algorithm which performs quickly and well with one random seed might never converge at all with a different one.

3.3 Kohonen Self-Organizing Map

A Kohonen SOM is a variation on a neural network which can be represented graphically as a two-dimensional grid of nodes. The grid is initialized to a set of random weight values. As the grid is exposed to data, each node reacts in some way to it. Nodes which have a strong response to a particular piece of data are strengthened such that they are encouraged to respond strongly to similar data in the future, while nodes with a weak response are weighted to keep their response to that type of data weak. In this way, different areas of the net are trained to fire a strong response to different sort of data. An unknown individual can be classified by sending it through the network and determining what area fires, and thus, what class the individual belongs to.

4. 0 Description

I used a simple character set for this preliminary testing: 15,000 samples of the digits 0 to 9, already scanned, quantized to two bits, de-skewed, and scaled. That sort of image pre-processing is out of the scope of this work.

The features I originally planned to use involve breaking the image data into 5x5 boxes and summing the number of 'dark' pixels in each box, breaking the image data into 4x5 boxes and summing the number of 'dark' pixels in each box, counting the number of dark pixels in each row, and in each column. As testing of the SOM began, I found the last method, counting the dark pixels in each column, performed so poorly it was not worth continuing to experiment with. I replaced it with another feature which was simply a concatenation of the vertical column data with the horizontal row data.

In the next step, each separate set of features was fed through K-Means, the genetic algorithm, and the Kohonen SOM, varying the parameters for each method in order to determine which conditions they work best under for this data.

5.0 Experimental Procedure

5.1 Kohonen SOM

I ran the 15,000 characters data set through each of four feature extractors: the 5x5 grid, the 4x5 grid, the row counter, and the combined row/column counter. The output from each of these was broken up into a training set of the first 5,000 samples, and a testing set of the last 10,000 samples. A smaller training suite of 1,000 samples was also pulled out of the bigger training set.

I used the SomPak Kohonen SOM package to implement the SOM. It consists of a set of command-line operated routines for building and training a Kohonen SOM. A description of the package and where it can be obtained can be found in the appendix. A series of script files was created for each feature set, each one carrying out the creation, training, and testing of a Kohonen SOM as well as extracting and summarizing the results of each run and creating a PostScript file of the SOM itself. Each of the four data sets, representing the four features, was tested on 32 different SOM's, each with different parameters. The parameters I chose to investigate were:

1. Training set size. I used both a set of 5,000 samples and a set of 1,000 samples to determine if the size of the training set had any appreciable affect on the accuracy of the network.

2. Number of training passes. This indicated how many times the training algorithm would cycle through the training data. I tested two settings: a combination of 1,000 passes through the first training cycle and 10,000 passes through the second training cycle, and a combination of 5,000 and 50,000 passes, respectively.

3. SOM shape. The Som Pak software allows for the creation of maps in which the 'nodes' are organized either hexagonally or in rectangles. The documentation that comes with Som Pak seems to indicate that a hexagonally organized map, with six connections for each node, might work better for some data sets than rectangular maps. I tried both.

4. Neighborhood function. This determines how the training neighborhood decreases in radius as training proceeds. The 'bubble' option is a step function; the other option is a gaussian decline.

5. SOM size. After some brief investigation, I determined SOM sizes around 10 or 15 units square seemed to provide the best results. I used a grid size of 10 units square and 15 units square for comparison's sake.

The steps involved in creating, training, testing, and evaluation the network are as follows:

1. Randomly initialize the network. SomPak include the executable 'randinit' for this purpose. Map size, shape, and neighborhood function are set at this time.

2. Training. SomPak includes the 'vsom' executable for this purpose. This was carried out in 2 stages, as per the recommendations in the SomPak documentation. The parameters to vsom include the number of training exemplars, the learning rate alpha, and the initial neighborhood radius. The neighborhood radius and alpha were constants though all the experiments, based on values available in the manual.

It should be noted that the parameter 'rlen', describing how many training exemplars the program should run through, is the method for specifying the number of training passes through the data. If the number specified by rlen is larger than the number of exemplars in the training data file, vsom iterates back through the file. For instance, when told to use 5000 training steps, but only given a file containing 1000 exemplars, vsom will iterate through that file 5 times.

The two training stages are an initial 'rough' training, during which rlen is small and alpha and the initial training radius are large. During the second phase, the map is fine tuned. rlen is usual much greater, while alpha and the neighborhood radius start out much smaller. In my experiments, I used an alpha value of .05 and an eight node radius in the first training stage, and an alpha of .02 and a three node radius in the second training stage.

3. Determine quantization error. This step 'computes the quantization error over all the samples in the data file'. It does not seem to have any effect on the actual output; it just reports a statistic concerning the performance of the map. This statistic does not quite correlate to accuracy.

4. Calibrate the map. This step is optional. Given a set of labeled training data, the program vcal will assign the labels to the appropriate nodes. This is primarily a time-saver, allowing later test samples to be labeled as they are classified by the network rather than having to do it by hand by comparing the coordinates of the node which fires in response to a sample with the clusters in the map. I did use this optional feature, and it was very helpful.

5. Test. During this stage, a batch of fresh, unlabeled data is presented to the network to be classified. The program 'visual', included for this purpose, returns the coordinates of the best-matching units for the sample, and the label associated with that unit, if any.

6. Extract and tally results. An awk script was used to extract the labels of the units each test data point was assigned by the som (if any.) I wrote a small c program which compared these results with the correct answers, keeping count of how many were correctly classified, how many were incorrectly classified, and how many were not classified at all. In the case of incorrect classifications, it also kept track of what the point was classified as opposed to what would have been correct.

7. Draw map. SomPak include a couple different programs for creating graphical representations of the maps. The program 'umat' was used to output the maps to PostScript files.

A sample script file for running the whole process to create one map can be found in the appendix.

5.2 K-Means Algorithm

As with the Kohonen SOM, the set of 15,000 characters was run through each of the four feature extractors: 5x5, 4x5, horizontal row counter, and the horizontal/vertical row counter. The vertical counter was discarded due to its very poor performance in the previous experiment. As the K-Means algorithm does not involve the same sort of training then testing cycle as the Kohonen SOM, the exemplars did not need to be split into groups.

I used Khoros, a very versatile information processing package with a wide range of capability. The routines it includes can be accessed either through the command line or a graphical user interface. I used the graphical user interface for this project. As with SomPak, information about this package and where it can be obtained is available in the appendix. A workspace was created in Khoros for each of four experiments per feature extractor, each excersizing a different combination of variable in the process. The parameters I chose to investigate were:

1. Data set size. I was curious as to whether the size of the data set presented to the k-means algorithm made any difference in its ability to perform. I ran both the full set of 15,000 characters and a set of the first 5,000 through the algorithm.

2. Initial cluster centers. The success of the K-Means algorithm is very sensitive to the choice of initial cluster centers. As it is basically a hill-climbing process, a poor choice of initial cluster centers can lead to a local maximum and a failure of the algorithm to perform to its full potential. Khoros allows initial cluster centers to be either chosen 'randomly', in which case it simply uses the first X data points in the data set for cluster centers, where X is the number of clusters, or else it allows he user to specify initial cluster centers in a separate data file. I ran it with both the 'random' cluster centers, and using specific intelligently chosen cluster centers. For specific cluster centers, I chose one exemplar of each digit, zero through nine. This did result in a certain amount of overlap in initial cluster centers between the two methods as the first ten data points in the file contained exemplars of most of the digits, and those were used as initial cluster centers in both types of experiments.

The process involved in running the K-Means algorithm is as follows:

1. Convert input data into the Khoros format. The Khoros routines are designed to deal with data presented in Khoros's own internal format. The application provides routines for converting ASCII files to a Khoros data object.

2. Run the algorithm. There are not many parameters for the K-Means algorithm as implemented here. The user can choose to specify initial cluster centers, and must list the number of clusters desired. The other parameter that could theoretically be changed is the distance measure used by the algorithm to calculate distance between cluster centers and data points. The Khoros K-Means implementation uses Euclidean distance for this. It does not have provisions for the use of other distance measures.

3. Convert the output. The results of the test must be changed from the Khoros data format back into plain text in order to use my extraction routines on them.

4. Tally results. I wrote a program which compares the K-Means output to the actual identity of each data point. Since the results are not calibrated as they were in the Kohonen SOM study, the correctness of the groupings is not at issue here so much as the actual composition of the clusters. In other words, the idea is not to determine if all the 5's have been correctly classified as 5's so much as it is to determine if all the 5's have been grouped together. To this end, I wrote a small C program which reported back the composition of each cluster. Sample results are listed below.

	Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6	Cluster 7	Cluster 8	Cluster 9
0's	0	1249	16	8	13	2	20	1	66	38
1's	0	0	4	0	2	4	21	988	3	207
2's	33	18	17	24	47	15	60	1	687	274
3's	10	3	652	4	0	26	24	2	979	51
4's	58	1	43	995	3	2	26	7	9	419
5's	2	26	100	25	27	1	24	0	220	291
6's	0	48	0	18	1477	0	11	38	133	74
7's	571	2	42	20	0	772	14	29	1	408
8's	46	39	141	72	3	1	694	26	60	598
9's	676	2	436	355	0	62	60	4	3	216

Figure 1. Sample of output (reformatted) from the results-tallying utility shows what was assigned to each cluster.

5.3 Genetic Algorithm

The two main questions anyone wanting to use a genetic algorithm must answer in regards to the problem at hand are: how will the data be represented in each individual string, and how will the fitness of each individual be calculated. For data representation, I used a variation on a scheme which proved successful in earlier work.

In a previous experiment in using genetic algorithms in clustering, I was attempting to cluster two groups of data. Each individual in the population consisted of an array of integers. The array had as many elements as there were data points to be classified. The population of individuals was created but randomly assigning a '0' or a '1' to each element in each individual. Each data point represented by a '0' in any particular individual was taken as belonging to one cluster, while each point represented by a '1' was placed in the other cluster. It is worth noting that using the representation scheme, each individual in the population was an attempt to cluster all the data points at once.

For this work, I used the same scheme, just modifying it to represent ten clusters rather than two. The elements were initialized randomly to a number from 0 to 9. For each feature vector, this number indicates which cluster the character represented by that element 'belongs' in. Note that the number given to a particular cluster in this scheme does not necessarily indicate what the characters in that cluster actually are. The numbers in the individual schemes are labels only; I could have just as easily used alphabetic characters or other symbols rather than digits in the individuals. The fact that a set of characters are clustered into group zero does not mean that they are all zeros. Actual character recognition is not part of this system.

An individual's fitness will be calculated by scanning the length of the individual, determining which elements are in the same group, accessing the feature sets referred to by the

elements, and calculating the maximum difference between corresponding places in the feature set in the set by comparing each feature of each character separately, noting the largest difference for each feature, and then adding those numbers together. The more fit individual will have a lower fitness rating.

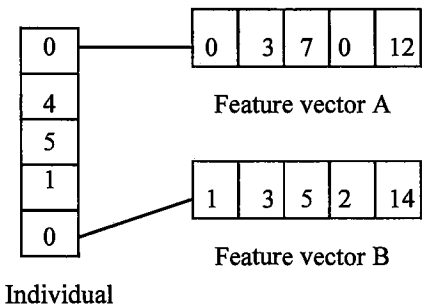


Figure 2: This sample individual places feature vectors A and B in the same cluster. The fitness of the individual is calculated by determining the maximum difference between each element of the vectors clustered together, and adding them all together. This classification would result in 7 (1+0+2+2+2) being added to the fitness value for the individual .

After running the genetic algorithm a few times, I found I had to make an adjustment to the fitness function. The way it was written, individuals were rewarded for minimizing the number of clusters they contained. This is clearly not the desired goal. I adjusted the fitness function by heavily penalizing any individual which excluded a cluster or did not assign featrue vectors to clusters in a nearly even distribution.

At first, individuals were 100 elements long, representing 100 characters to be clustered at a time. I planned to use longer individuals and cluster more characters per run of the genetic algorithm one I was sure the system was functioning properly; however, it ran so slowly with just 100 elements in each individual that I decided experimentation with longer individuals was not feasible.

The results from the genetic algorithm's classification were run through the same cluster counter program as was used to the K-Means results.

6.0 Results and Conclusions

6.1 Kohonen SOM

The goal of this part of the project was to determine if a Kohonen SOM could cluster handwritten character data at all, how well, how each factor (feature set, map size, and so on) affected the overall success, and what combination of factors worked best.

The parameters were found to have the following effects on the accuracy of the SOM:

1. Training set size. The maps trained with a small training set tended to have a slightly better response than those trained on a large data under most conditions. Other factors, however, had a much larger effect on the overall accuracy of the SOM.

2. Number of training passes. The number of passes through the training data was found to have a large effect on the success of the network. Maps trained with many iterations specified with the rlen parameter did significantly better than those trained with few cycles.

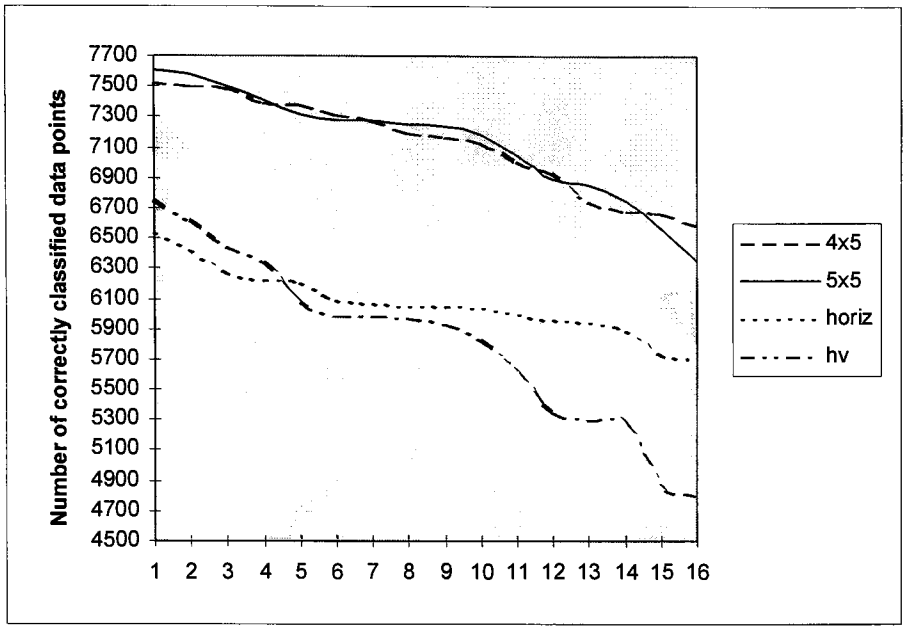
3. SOM shape. The map geometry, hexagonal or rectangular, did not have a large impact on the accuracy of the process.

4. Neighborhood function. The 'bubble' neighborhood function was found to perform significantly better than the gaussian decline. This is nice because the bubble function also ran quite a bit faster.

5. Map size. The larger, 15x15 unit map was found to produce, in general, more correctly classified data points than the 10x10 map, although the larger map also resulted in more unclassified units. Considering that an unclassified unit is, for these purposes, no better or worse than an incorrectly classified one, the larger map is better.

6. Feature set. Of the four feature sets used, the 5x5 box feature had the highest average percent correct answers, followed closely by the 4x5 box set. The horizontal data set and the combined horizontal-vertical feature sets both were around 10 percent less accurate.

The different feature sets also had different sensitivity to the other parameters. The horizontal data set showed the least sensitivity to changes in other parameters, with the 5x5 and 4x5 box feature sets only slightly more sensitive. The horizontal-vertical combination feature set was very sensitive to the variations in other parameters. A graph below illustrated this point.



While the best results obtained were only around 75 percent correct, I still count this as a successful clustering method for this purpose. Some OCR applications do achieve recognition rates of 95 percent or better, and this is clearly much less accurate. However, it is not intended to be a complete recognition system but simply an aid to a human operator working to classify and label difficult examples. Also, the characters I am working with are difficult examples. Handwritten text like I am using is much less regular than printed characters, and the preprocessing on my data set is not the most exhaustive -- there are exemplars which are not scaled across the entire 20x30 pixel grid, but that 'float' in one corner or another.

In short, I feel that there's enough potential for success here that more investigation in this method is warranted. The work could be repeated with a different data set, or other parameters could be tested. Also, some way of adding a 'confidence' measure to the output would be helpful, as in, reporting back that the SOM has classified a particular character to a cluster with a certain percentage confidence that the answer is correct. While this was included in the original scope of this project, no way to incorporate this idea into the operation of SomPak was readily available.

6.2 K-Means Algorithm

The goal of this experiment was to determine how well the k-means algorithm can cluster handwritten character data and how each parameters to the algorithm (feature set, data set size, and initial cluster center) affected its performance.

The performance of the algorithm can be judged according to speed of execution and tightness of the clusters generated -- the goal is to have the all the same characters in each cluster.

The number of iterations it takes the algorithm to converge is an indicator of how fast it runs. There did not seem to be a great deal of correlation between any of the studied factors and the number of iterations performed. On average, the small data set with intelligently chosen initial cluster centers converged in the fewest number of iterations, but that is the only clear indication in the data. As the algorithm did not require really large amounts of time to run in any of the tested cases anyway, the speed does not have to be a concern.

As for the clustering quality, there was a surprisingly small difference in the quality of the clusters produced with random initial cluster centers vs. specific initial cluster centers. While the clustering was slightly better with pre-chosen cluster centers, the difference was small. This can possibly be attributed to the fact that most of the initial cluster centers were actually the same between the two.

The largest difference in quality came from the feature sets chosen. As with the Kohonen SOM, the 5x5 and 4x5 feature sets performed best, followed by the horiz set, with the horizontal-vertical set giving the poorest results of the four. Something worth noting is that some characters clustered much better than others. 0's, 1's, and 6's clustered best, while the algorithm seemed to have the most trouble classifying 3's, 8's and 9's. It is possible that this is an effect of the cluster centers chosen.

While this study does show some success using this method for clustering character data, it does not seem to behave as well as the Kohonen SOM for this purpose. More experimentation is called for, perhaps using different distance measures (Mahalanobis distance, for example, has been found to perform better in some applications) and working more with the idea of different initial cluster centers.

6.3 Genetic Algorithm

The first thing noticed about the genetic algorithm was that with this (admittedly expensive) fitness function, it was substantially slower than either of the other two methods. In fact, it was slow enough (taking on average roughly an hour and a half to complete one run) that it might almost be faster to go ahead and classify samples by hand rather than use a genetic algorithm configured as it was for this experiment.

In addition, the genetic algorithm was not nearly as effective at performing the classification as either of the other two methods investigated. Had the algorithm been run on more than 100 samples at a time, thus having more data to work with, it might have been more successful. As it was with so few samples, any error or uncertainty in the clustering becomes very significant.

6.4 Comparisons

Both the Kohonen map and the K-means algorithm showed encouraging results. They both clustered large numbers of characters quickly and with enough success to greatly speed the

process of labeling character data. The performance of the genetic algorithm, on the other hand, was quite poor in comparison to the other two methods.

As for the feature sets, the 4x5 and 5x5 features returned the most tightly clustered results under both the Kohonen SOM and K-Means. With the genetic algorithm, all feature sets behaved equally poorly.

6.5 Suggestions For Further Study

It is clear that should a genetic algorithm ever be used for this purpose, more work must be done to make it effective. The problems I encountered using the genetic algorithm, slowness and inaccuracy, are both attributable to the choice of fitness function. There is plenty of room for work on finding a better fitness function for this problem.

One thought would be to use the unfiltered character data in place of the extracted feature vectors, calculating the fitness from hamming distance between the characters. I did not attempt this method because part of my goal in this project was to test the different features themselves in each clustering method.

More work should be done with the K-means algorithm; specifically, other distances measures besides the one used by Khoros should be investigated.

A major factor which would be useful to add to this research in moving towards a final application would be finding a way to include, along with assigning each character to a cluster, a confidence level to that grouping. For instance, a certain character could be included in a cluster with a 70% confidence level. This feature is available to some extent in a variation on the K-Means algorithm known as 'fuzzy K-means'. It could also be added to a Kohonen SOM, perhaps as a function of the distance the node activated by a certain character from the geographical center of the cluster in the SOM. Adding this functionality to a GA, however, would require much more work.

The point of doing this would be so that in a final application, the entries in a particular cluster could be presented to the user in a ranked order, making it quicker to discard incorrectly classified entries.

Even without further investigation in these areas, a full character classification system could be written now which would be of value in creating labeled character data. While a lot of room for optimizing the system remains, one could be put together based on this work which would serve the purpose well.

7.0 Appendixes

7.1 Kohonen SOM Example Script

```
echo initializing map
./randinit -din 5x5.dat -cout 5x5_b_f_r_b.map -xdim 15 -ydim 15 -topol rect -neigh
bubble
echo first training run
./vsom -din 5x5.dat -cin 5x5_b_f_r_b.map -cout 5x5_b_f_r_b.map -rlen 1000 -alpha 0.05
-radius 8
echo second training run
./vsom -din 5x5.dat -cin 5x5_b_f_r_b.map -cout 5x5_b_f_r_b.map -rlen 10000 -alpha
0.02 -radius 3
echo quantizatin error
./qerror -din 5x5.dat -cin 5x5_b_f_r_b.map
echo calibrating map
./vcal -din 5x5.calib -cin 5x5_b_f_r_b.map -cout 5x5_b_f_r_b.map
echo producing output
./visual -din 5x5.test -cin 5x5_b_f_r_b.map -dout 5x5_b_f_r_b.out
./strip 5x5_b_f_r_b.out > 5x5_b_f_r_b.strip
echo tallying results
./percent 5x5.check 5x5_b_f_r_b.strip > 5x5_b_f_r_b.results
echo drawing ps map
./umat -cin 5x5_b_f_r_b.map -ps1 > 5x5_b_f_r_b.ps
```

7.2 Kohonen SOM Drawing

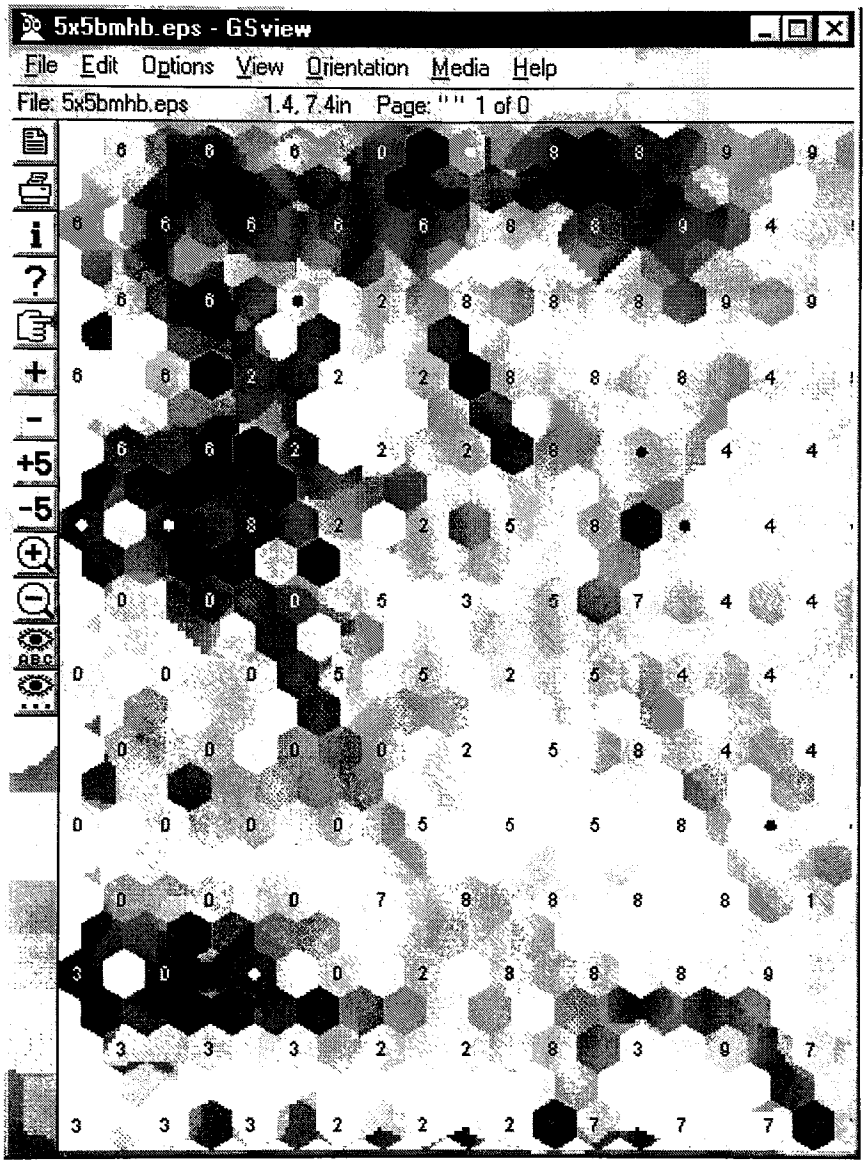


Figure 3. A drawing of the graphical representatin of a Kohonen Self-Organizing Map as created by SomPak.

7.3 Background Material

K-Means Algorithm:

Class notes from the graduate imaging science course SIMG-784, Spatial Pattern Recognition, as taught in the spring quarter of 1995 by Dr. Harvey Rhody.

Genetic Algorithms:

Class notes and experiments from the graduate computer science course ICSG-756, Genetic Algorithms, as taught in the spring quarter of 1995 by Dr. Peter Anderson.

Kohonen SOM:

Class notes and experiments from the graduate computer science course ICSG-755, Neural Network Seminar, as taught in the winter quarter of 1994 by Roger Gaborski.

Charatcter Recognition/Feature Selection:

Gader, Paul et al. "Matching Database Records to Handwritten Text". SPIE Proceedings on Document Recognition. Vol. 2181. SPIE Publications, 1994. 67-75.

Grother, Patrick J. "Cross Validation Comparison of NIST OCR Databases". SPIE Proceedings on Character Recognition Technologies. Vol. 1906. SPIE Publications, 1993. 296-306.

Wilson, CL. "Efficetiveness of Feature and Classifier Algorithms in Character Recognition Systems". SPIE Proceedings on Character Recognition Technologies. Vol. 1906. SPIE Publications, 1993. 255-265.

Garris, Michael D., Jon Geist, and R.Allen Wilkenson. "Machine-Assisted Human Classification of Segmented Characters For OCR Testing and Training". SPIE Proceedings on Character Recognition Technologies. Vol. 1906. SPIE Publications, 1993. 208-271.

7.4 Other Sources

K-Means Algorithm:

The Khoros Pro package was used for the k-means algorithm. It is a UNIX-based software package created for data visualization and manipulation, espeically concerning image processing problems of various types. The package itself (including binaries for a variety of systems), along with a users manual, can be purchased from Khoral Research Inc., or else the source code can be downloaded from their web site, **www.khoral.com**.

Kohonen SOM:

Som_Pak is a set of routines for the creation, training, and use of a Kohonen SOM. Version 3.1 was used for these experimenst. The package is written for UNIX and MS-DOS systems by the SOM Programming Team of the Helsinki University of Technology Laboratory of Computer and Information Science. The program can be downloaded from **http://www.cis.hut.fi/nnrc/nnrc-programs.html**. More general information is available from **http://www.cis.hut.fi/~hynde/lvq/**.

7.5 Percentage Correct Calculation Code

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
```

```
{
        FILE *orig, *output;
        int right, wrong, total, noclass, i, j, tally;
        float percent;
        char orig_line[3], out_line[3];
        int matrix[10][10];

        right = wrong = noclass = total = tally = 0;

        for(i = 0; i < 10; i++)
                for(j = 0; j < 10; j++)
                        matrix[i][j] = 0;

        if(argc !=3)
        {
                printf("USAGE: percent <check file> <nn output file>\n");
                exit(0);
        }

        orig = fopen(argv[1], "r");
        output = fopen(argv[2], "r");

        fgets(orig_line, 4, orig);
        fgets(out_line, 4, output);

        while((!feof(orig)) && (!feof(output)))
        {
                total++;
                if (isspace(out_line[0]))
                        noclass++;
                else if(orig_line[0] == out_line[0])
                        right++;
                else
                {
                        matrix[atoi(orig_line)][atoi(out_line)]++;
                        wrong++;
                }

                fgets(orig_line, 4, orig);
                fgets(out_line, 4, output);
        }

        fclose(orig);
        fclose(output);
```

```
        printf("Number correct: %d\n", right);
        printf("Number wrong: %d\n", wrong);
        printf("Number unclassified: %d\n", noclass);
        printf("Total samples: %d\n", total);
        printf("Percent correct: %f\n", (((float) right/(float) total)*100));
        printf("Percent incorrect: %f\n", (((float) wrong/(float) total)*100));
        printf("Percent not classified: %f\n", (((float) noclass/(float) total)*100));
        printf("Error Matrix: \n   0 1 2 3 4 5 6 7 8 9\n");
        for(i = 0; i < 10; i++)
        {
                printf("%d: ", i);
                for(j = 0; j < 10; j++)
                {
                        printf("%d ",matrix[i][j]);
                        tally += matrix[i][j];
                }
                printf("\n");
        }
        printf("Tally: %d\n", tally);
}
```

7.6 Cluster Composition Code

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int group0[10];
        int group1[10];
        int group2[10];
        int group3[10];
        int group4[10];
        int group5[10];
        int group6[10];
        int group7[10];
        int group8[10];
        int group9[10];

        int i;

        FILE *check;
        FILE *output;

        char orig_line[10], out_line[10];
```

```
if (argc != 3)
        exit(0);

for(i = 0;i<10;i++)
{
        group0[i] = 0;
        group1[i] = 0;
        group2[i] = 0;
        group3[i] = 0;
        group4[i] = 0;
        group5[i] = 0;
        group6[i] = 0;
        group7[i] = 0;
        group8[i] = 0;
        group9[i] = 0;
}

check = fopen(argv[1], "r");
output = fopen(argv[2], "r");

fgets(orig_line, 9, check);
fgets(out_line, 9, output);

while((!feof(check)) && (!feof(output)))
{
        switch(out_line[0])
        {
                case '0':
                        group0[atoi(orig_line)]++;
                        break;
                case '1':
                        group1[atoi(orig_line)]++;
                        break;
                case '2':
                        group2[atoi(orig_line)]++;
                        break;
                case '3':
                        group3[atoi(orig_line)]++;
                        break;
                case '4':
                        group4[atoi(orig_line)]++;
                        break;
                case '5':
                        group5[atoi(orig_line)]++;
                        break;
```

```
                case '6':
                        group6[atoi(orig_line)]++;
                        break;
                case '7':
                        group7[atoi(orig_line)]++;
                        break;
                case '8':
                        group8[atoi(orig_line)]++;
                        break;
                case '9':
                        group9[atoi(orig_line)]++;
                        break;
        }
        fgets(orig_line, 9, check);
        fgets(out_line, 9, output);
}

fclose(check);
fclose(output);

printf("Cluster 0:\n");
for(i = 0; i<10;i++)
        printf("\t%d's: %d\n", i, group0[i]);

printf("\nCluster 1:\n");
for(i = 0;i<10;i++)
        printf("\t%d's: %d\n", i, group1[i]);

printf("\nCluster 2:\n");
for(i = 0;i<10;i++)
        printf("\t%d's: %d\n", i, group2[i]);

printf("\nCluster 3:\n");
for(i = 0;i<10;i++)
        printf("\t%d's: %d\n", i, group3[i]);

printf("\nCluster 4:\n");
        for(i = 0;i<10;i++)
        printf("\t%d's: %d\n", i, group4[i]);

printf("\nCluster 5:\n");
for(i = 0;i<10;i++)
        printf("\t%d's: %d\n", i, group5[i]);

printf("\nCluster 6:\n");
```

```
        for(i = 0;i<10;i++)
                printf("\t%d's: %d\n", i, group6[i]);

        printf("\nCluster 7:\n");
        for(i = 0;i<10;i++)
                printf("\t%d's: %d\n", i, group7[i]);

        printf("\nCluster 8:\n");
        for(i = 0;i<10;i++)
                printf("\t%d's: %d\n", i, group8[i]);

        printf("\nCluster 9:\n");
        for(i = 0;i<10;i++)
                printf("\t%d's: %d\n", i, group9[i]);
}
```

7.7 Genetic Algorithm Code Modifications

The genetic algorithm I used was based on C code provided by Dr. Peter Andersen. I made modifications in the fitness function and the data representation. The functions containing my work are printed here.

```
typedef struct {
        int feature[FEATURE_LENGTH];
} point;

unsigned char **p;
int     *perm_help, *r, *c;
int     **M;
float MAX_HERO;
float fitness[1000];
float hero;

int     individual_length;
int mut_count, trial;

point data [100];
int group[100];

/*=================================================================*/
main( argc, argv ) int argc; char **argv;
{
        int who, c1, c2, p1, p2, a,b,c,d, tmp, perm_max;
        int j = 0;
        int k;
```

```
        FILE *fp;
        char temp[200];
        char *shorttemp;

        fp = fopen("data", "r");

        for(j = 0; j < 100; j++)
        {
                fgets(temp, 200, fp);
                shorttemp = strtok(temp, " ");
                data[j].feature[0] = atoi(shorttemp);
                for(k = 1; k < FEATURE_LENGTH; k++)
                {
                        shorttemp = strtok(0, " ");
                        data[j].feature[k] = atoi(shorttemp);
                }
        }
        fclose(fp);

        params( argc, argv );
        init();

        forall( who, 0, POP_SIZE-1 ) fitness[who] = fv(who);

        forall( trial, 1, LOOPS ) {
                perm_max = POP_SIZE;

#define GET(A) tmp = random( perm_max );\
                                        A = perm_help[tmp];\
                                        swap( tmp, perm_max-1 );\
                                        perm_max--;

                if( two_parents ) {
                        GET(a);
                        GET(b);
                        GET(c);
                        GET(d);

                        if( fitness[a] < fitness[b] ) {
                                c1 = a;
                                p1 = b;
                        } else {
                                c1 = b;
                                p1 = a;
                        }
```

```
                        if( fitness[c] < fitness[d] ) {
                                c2 = c;
                                p2 = d;
                        } else {
                                c2 = d;
                                p2 = c;
                        }
                        if( hero >= MAX_HERO ) {
                                int i, j;

                                break;
                        }
                        make_children( p1, p2, c1, c2 );
                        if( MUT_RATE > 0.0 ) {
                                mutate( c1 );
                                mutate( c2 );
                        }
                        fitness[c1] = fv(c1);
                        if(  hero >= MAX_HERO ) {
                                printf( "Maximum hero reached: %d\n", hero );
                                exit( 0 );
                        }
                        fitness[c2] = fv(c2);
                        if(  hero >= MAX_HERO ) {
                                printf( "Maximum hero reached: %d\n", hero );
                                exit( 0 );
                        }
                }
        }
        printf( "\t\t\tStopped after %d trials. Hero = %f\n", trial, hero );

        if(  hero >= MAX_HERO ) {
                printf( "Maximum hero reached: %d\n", hero );
                exit( 0 );
        }

}

int fv( who ) int who; /* fitness value of individual who */
#define VV(X)           (2*(X)-1)
{
        static  int     count = 0;
        int     I, J, K, i, j, k, l;
        int ones[50], twos[50], threes[50], fours[50], fives[50];
        int sixes[50], sevens[50], eights[50], nines[50], zeros[50];
```

```
int count0, count1, count2, count3, count4;
int count5, count6, count7, count8, count9;
int the_fitness;
int max, min;

count++;

the_fitness =  0;
count0 = count1 = count2 = count3 = count4 = 0;
count5 = count6 = count7 = count8 = count9 = 0;

for( i = 0; i<100; i++)
{
        switch(p[who][i])
        {
                case 0:
                        zeros[count0] = i;
                        count0++;
                        break;
                case 1:
                        ones[count1] = i;
                        count1++;
                        break;
                case 2:
                        twos[count2] = i;
                        count2++;
                        break;
                case 3:
                        threes[count3] = i;
                        count3++;
                        break;
                case 4:
                        fours[count4] = i;
                        count4++;
                        break;
                case 5:
                        fives[count5] = i;
                        count5++;
                        break;
                case 6:
                        sixes[count6] = i;
                        count6++;
                        break;
                case 7:
                        sevens[count7] = i;
```

```
                        count7++;
                        break;
                case 8:
                        eights[count8] = i;
                        count8++;
                        break;
                case 9:
                        nines[count9] = i;
                        count9++;
                        break;
        }
}

the_fitness = 0;

for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count0 != 0)
        {
                max = data[zeros[0]].feature[i];
                min = data[zeros[0]].feature[i];
                for(j = 1; j < count0; j++)
                {
                        if(data[zeros[j]].feature[i] > max)
                                max = data[zeros[j]].feature[i];
                        if(data[zeros[j]].feature[i] < min)
                                min = data[zeros[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count1 != 0)
        {
                max = data[ones[0]].feature[i];
                min = data[ones[0]].feature[i];
                for(j = 1; j < count1; j++)
                {
                        if(data[ones[j]].feature[i] > max)
                                max = data[ones[j]].feature[i];
                        if(data[ones[j]].feature[i] < min)
                                min = data[ones[j]].feature[i];
                }
                the_fitness += (max - min);
```

```
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count2 !=0)
        {
                max = data[twos[0]].feature[i];
                min = data[twos[0]].feature[i];
                for(j = 1; j < count2; j++)
                {
                        if(data[twos[j]].feature[i] > max)
                                max = data[twos[j]].feature[i];
                        if(data[twos[j]].feature[i] < min)
                                min = data[twos[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count3 != 0)
        {
                max = data[threes[0]].feature[i];
                min = data[threes[0]].feature[i];
                for(j = 1; j < count3; j++)
                {
                        if(data[threes[j]].feature[i] > max)
                                max = data[threes[j]].feature[i];
                        if(data[threes[j]].feature[i] < min)
                                min = data[threes[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count4 != 0)
        {
                max = data[fours[0]].feature[i];
                min = data[fours[0]].feature[i];
                for(j = 1; j < count4; j++)
                {
                        if(data[fours[j]].feature[i] > max)
                                max = data[fours[j]].feature[i];
                        if(data[fours[j]].feature[i] < min)
```

```
                        min = data[fours[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count5 != 0)
        {
                max = data[fives[0]].feature[i];
                min = data[fives[0]].feature[i];
                for(j = 1; j < count5; j++)
                {
                        if(data[fives[j]].feature[i] > max)
                                max = data[fives[j]].feature[i];
                        if(data[fives[j]].feature[i] < min)
                                min = data[fives[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count6 != 0)
        {
                max = data[sixes[0]].feature[i];
                min = data[sixes[0]].feature[i];
                for(j = 1; j < count6; j++)
                {
                        if(data[sixes[j]].feature[i] > max)
                                max = data[sixes[j]].feature[i];
                        if(data[sixes[j]].feature[i] < min)
                                min = data[sixes[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count7 != 0)
        {
                max = data[sevens[0]].feature[i];
                min = data[sevens[0]].feature[i];
                for(j = 1; j < count7; j++)
                {
```

```
                        if(data[sevens[j]].feature[i] > max)
                                max = data[sevens[j]].feature[i];
                        if(data[sevens[j]].feature[i] < min)
                                min = data[sevens[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count8 != 0)
        {
                max = data[eights[0]].feature[i];
                min = data[eights[0]].feature[i];
                for(j = 1; j < count8; j++)
                {
                        if(data[eights[j]].feature[i] > max)
                                max = data[eights[j]].feature[i];
                        if(data[eights[j]].feature[i] < min)
                                min = data[eights[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}
for(i = 0; i < FEATURE_LENGTH; i++)
{
        if(count9 != 0)
        {
                max = data[nines[0]].feature[i];
                min = data[nines[0]].feature[i];
                for(j = 1; j < count9; j++)
                {
                        if(data[nines[j]].feature[i] > max)
                                max = data[nines[j]].feature[i];
                        if(data[nines[j]].feature[i] < min)
                                min = data[nines[j]].feature[i];
                }
                the_fitness += (max - min);
        }
}

if(count0 > 13 || count0 < 8)
        the_fitness += 2000;
if(count1 > 13 || count1 < 8)
        the_fitness += 2000;
```

```
        if(count2 > 13 || count2 < 8)
                the_fitness += 2000;
        if(count3 > 13 || count3 < 8)
                the_fitness += 2000;
        if(count4 > 13 || count4 < 8)
                the_fitness += 2000;
        if(count5 > 13 || count5 < 8)
                the_fitness += 2000;
        if(count6 > 13 || count6 < 8)
                the_fitness += 2000;
        if(count7 > 13 || count7 < 8)
                the_fitness += 2000;
        if(count8 > 13 || count8 < 8)
                the_fitness += 2000;
        if(count9 > 13 || count9 < 8)
                the_fitness += 2000;

        the_fitness = -the_fitness;
        if(the_fitness == 0)
                the_fitness = -10000;

        if( print_every_fitness )
        {
                printf( "%4d fitness: %d\n", count, the_fitness );
                for(j = 0; j<100; j++)
                        printf("%d", p[who][j]);
                putchar('\n');
        }
        if( the_fitness > hero ) {
                hero = the_fitness;
                if( print_every_hero ){
                        printf( "New hero %4d %d\n", count, the_fitness);
                        for(j = 0; j<100; j++)
                                printf("%d\n", p[who][j]);
                }
        }
        if ( hero >= MAX_HERO ) {

                printf( "Maximum hero reached: %d, fitness evaluation: %d\n",
                        hero, count );
        }
        return( the_fitness );
}
```