Preface: Data

All data for this project was downloaded from the publicly accessible site: http://data.un.org. Our data sets are taken from the 2000-2005 World Population Prospects (United Nations Population Division). For your convenience, I have attached the data to this report. All attachments are attached in order of first mention in this report.

In order to explore the algorithms presented in class, we chose an imperfect, yet sufficiently large and interesting, data set that is publicly available. Our data examines seven features of member UN countries: net migration, deaths per year, birth rate, life expectancy, population change, fertility, and infant mortality rate. The UN classifies its members into 6 regions: Africa, Asia, Europe, Latin America, Oceania, and North America. We wanted to know if it was possible to determine each country's region solely on the 7 demographic features we selected. If this were possible, then there would be non-geographic and somewhat non-cultural support for the UN's classification scheme.

We answer each question by using either the whole data set or narrowing it down to a couple features or classes. We divided our data in half alphabetically such that each class was represented equally in the training set and the test set. The name of a country, should not, a priori, effect the demographic data, therefore this is a valid split equivalent to random selection.

1) $J(w) = \left(w^T S_B w\right) \div \left(w^T S_W w\right)$ **vs** $J(w) = \left(w^T S_B w\right)$

For this question, we looked at three groups of data. We looked at how this classifier dealt with two of our classes that were close to one another (Africa and Asia) and two that were further apart (Africa and Europe) with respect to the birth rate and life expectancy feature vectors. Lastly, as a control, we generated random Gaussian data of the same size where class 1 had a mean of (-8, -8) and class 2 had a mean of (8, 8) and the variance was 10. We used the same randomly generated data for both classifiers. The code for this section is attached at the end of this document as "ECE662HW2Q1.m."

Results in tabular form and graphical:

Table 1:

	Misclassification	Africa vs Europe	Africa vs Asia	Gaussian Data
$J(w) = (w^T S_B w) \div (w^T S_W w)$	Class 1 %	25.93	14.81	22.22
	Class 2 %	0	56	10
$J(w) = (w^T S_B w)$	Class 1 %	18.52	22.22	33.33
	Class 2 %	0	20	15

Figure 1: Africa vs Europe $J(w) = (w^T S_B w)$

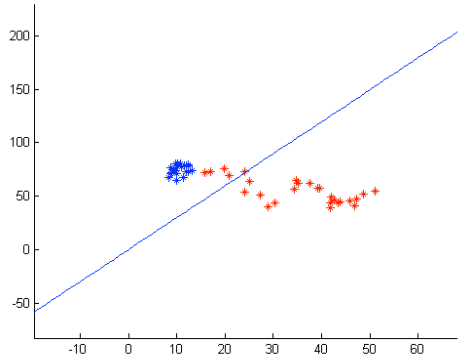


Figure 2: Africa vs Asia $J(w) = (w^T S_B w)$

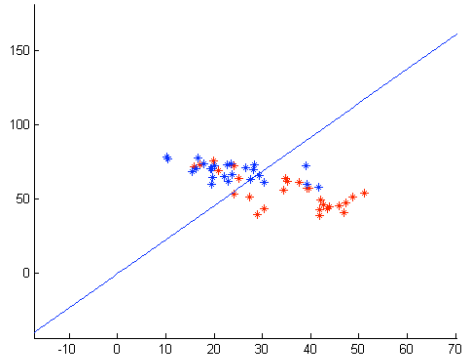


Figure 3: Gaussian Data $J(w) = (w^T S_B w)$

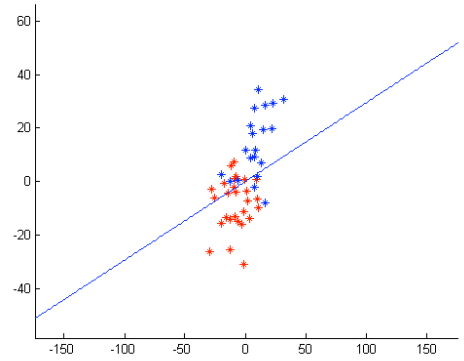


Figure 4: Africa vs Europe

$J(w) = (w^T S_B w) \div (w^T S_W w)$

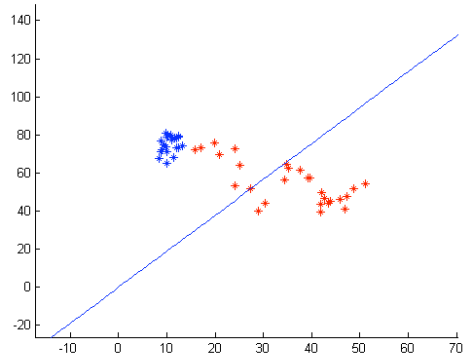


Figure 5: Africa vs Asia

$J(w) = (w^T S_B w) \div (w^T S_W w)$

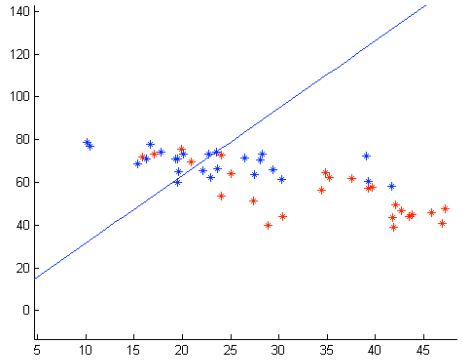
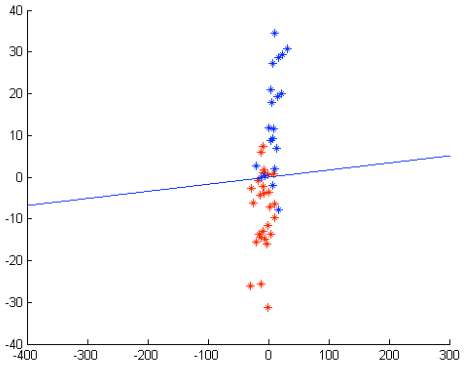


Figure 6: Gaussian Data

$J(w) = (w^T S_B w) \div (w^T S_W w)$



We expected that the optimal value of w to be the one we found in class: $w_0 = S_W^{-1}(mean_1 - mean_2)$. The results painted a much murkier picture. While it did better than the value optimized for $J(w) = (w^T S_B w)$: $w^T_{alt} = [-1, mean_1(1) - mean_2(1) / mean_1(2) - mean_2(2)]$ in the Gaussian data, w_0 only beat w_{alt} once in our region classification data. Therefore, it seems likely that, based on our numerical results, w_0 is only likely to be the optimal value of w when the data is normal.

2.

a) Artificial Neural Network

In order to take advantage of a state-of-the-art ANN, we chose to use an open-source utility called FANN (http://leenissen.dk/fann/). After some experimentation with the different language bindings, we chose the Visual Studio bindings and modified the sample code (attached at the end of this report as "xor_train.c") to use our data.

Our neural network was setup to include seven different inputs, one for each feature vector. We used one hidden layer and six outputs, one for each class. Having six outputs allowed the neural network to "choose" a class by setting one of the six outputs close to 1 and the others close to -1, with each class corresponding to one of the outputs. The neural network algorithm determined the weights for the arcs between each of the

layers through back-propagation, which bases the new arc weights on the results of the success or failure of the classification. We limited our algorithm to 10,000 iterations (or an error rate of less that 10%) and found that the algorithm achieved a steady state with respect to error rate for our training data well before it hit the 10,000th iteration in each trial.

With Artificial Neural Networks the functions (always of one variable) at each node as well as the number of hidden nodes (and layers) are fixed beforehand. While the default values suggested by the sample code were probably 'okay', we decided to vary both the number of hidden nodes and their function to see if we could find better values. Our results, for the test data, in terms of mean-squared error are displayed below:

Table 2:

f(x)	**1 / (1 + e^-x)**	**Gaussian**	**1 / (1 + e^-x)**	**1 / (1 + e^-x)**
# of Nodes	**3**	**3**	**1**	**10**
Mean-squared Error	11.531%	13.0547%	11.4522%	11.8255%

From the results, it is clear that a higher number of nodes has very little, and possibly bad, effect on the error. This may seem counter-intuitive at first because we are making the classifier more optimized. However, the key here is that we are allowing the neural network to *overfit* the training data by adding more nodes, which makes the error lower for the training data, but can increase the error for the test data. This bears out in the results for the training data: at ten nodes we achieved 10.1% error, at three it was 10.5%, and at one it was 11.1%. The Sigmoid function did do noticeably, if not hugely, better than the Gaussian function, which is why we only tested it for one node count.

b) Support Vector Machine

Just as we had for the ANN, we opted to utilize an open-source program for our SVM implementation. We chose PyML (http://pyml.sourceforge.net/), which is a machine learning library for the Python programming language. It implements a support vector machine algorithm for python called libsvm. Our code, utilizing the PyML library functions, is attached at the end of the document as "svm.py."

An individual SVM will only create a single decision hyper surface to select between two classes. However, they may be used to classify data among many classes if they are used in conjunction with one another. For our data, we could use an SVM to separate the data into groups of three, then for each of those groups into a single class and a set of two classes and finally separate that last group into two classes. The downside to doing this is that errors propagate down the tree. A misclassification in the first step guarantees failure. Therefore, the results will never be more precise than when we only look at separating two classes. For brevity, we have considered only this lower bound on the error. Below are the SVM results for classifying a country as either in the Africa or Asia regions and either Africa or Europe based on *all* features.

Figure 7: Africa vs. Asia:

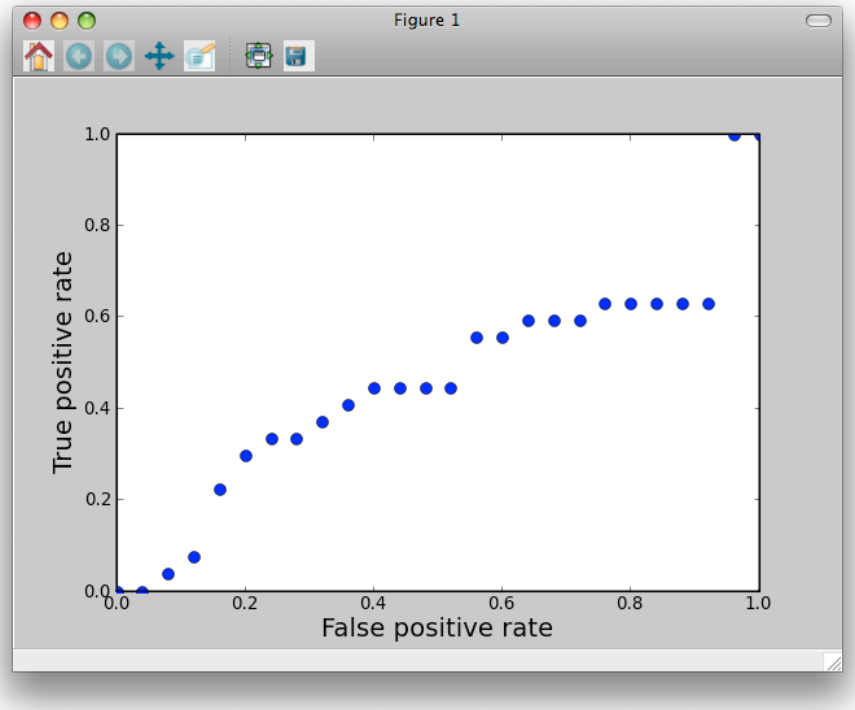
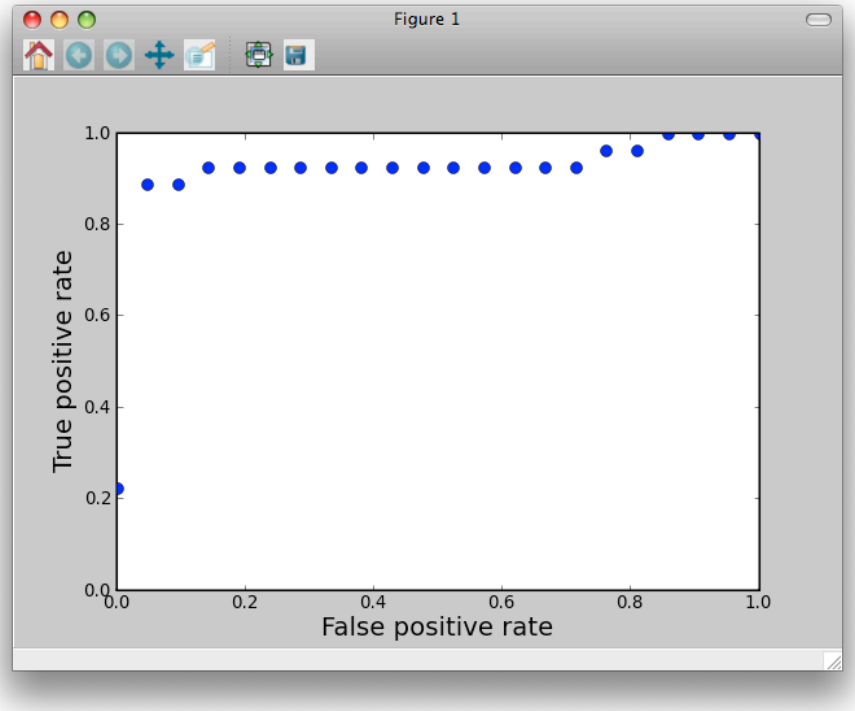


Figure 8: Africa vs. Europe:



As expected, because the European and African data is further apart than the African and Asian data the classification is much better. It is alarming at first that the Africa versus Asia classification seems to do worse than the classifier in question 1. However, this is actually expected. The African and Asian data sets were so close that for question 1 we used two of the most defining features for the two. However, with the SVM, we are considering all of the feature vectors equally. We are, in a sense, confusing the SVM by adding messy data.

c) Comparison of ANN and SVM

While our SVM does just as well as the ANN for well separated data, as seen in 'Figure 8', It is not only harder to implement for the full six-way classification of region, but also more error prone. As shown in 'Figure 7', the SVM barely does better than a simple guess when the data is clumped. Compare that to the results for the NN, where even accounting for the other data sets, the error is still decently low. However, because

we did not perform a direct comparison, we do not have enough data to say for sure that the ANN is more accurate.

3.

For this section feature vectors have been referenced by their ids: 2-net migration, 3-deaths per year, 4-birth rate, 5-life expectancy, 6-population change, 7-fertility, and 8-infant mortality rate.

a) Parzen Windows

For the Parzen windows method, we chose to use a Gaussian window. We wanted to find the optimal variance for our window based on several feature vector pairs where each vector's partner is the most helpful. We looked at this data for linear, Manhattan, Euclidian and 100-norm distance measures. The table below (Table 3) shows the results we obtained. The code used to determine these values is attached as "parzwin.m" at the end of this file. From the results, it appears that vectors 7 and 4 with a variance of 3 obtain the best errors.

Table 3: Parzen Window Results:

Distance Measure	Vector(s)	Variance	% Error
1D	4	3	40.82
Manhattan	2, 8	8	55.1
	3, 8	17	61.22
	4, 6	3	63.27
	5, 4	10	74.49
	6, 8	12	56.12
	7, 4	3	41.84
	8, 4	7	44.9

Distance Measure	Vector(s)	Variance	% Error
Euclidean	2, 4	15	44.9
	3, 8	8	54.08
	4, 5	15	35.71
	5, 4	5	35.71
	6, 4	12	47.96
	7, 4	3	40.82
	8, 4	2	41.84
100-norm	2,7	1	61.22
	3,4	11	60.2
	4,8	19	51.02
	5,4	4	36.73
	6,8	6	50
	7,4	3	40.82
	8,4	4	42.86

b) K-NN

For the K-Nearest Neighbor method, we wanted to find the optimal K value based on several feature vector pairs where each vector's partner is the most helpful. We looked at this data for linear, Manhattan, Euclidian and 100-norm distance measures. The table below (Table 4) shows the results we obtained. The code used to determine these values is attached as "knearn.m" at the end of this file. These results are a bit less conclusive than the Parzen window method's, but vectors 4, 5, and 7 and values of K between 12 and 30 (admittedly a wide range) seem to do well.

Table 4: K-Nearest Neighbor Results

Distance Measure	Vector(s)	K	% Error
1D	7	12	37.76
Manhattan	2,8	25	50
	3,2	15	61.27
	4,7	30	36.73
	5,4	16	67.35
	6,4	9	51.35
	7,4	30	36.73
	8,4	24	43.88
Euclidean	2,8	14	45.92
	3,6	4	51.02
	4,5	14	38.78
	5,4	16	35.71
	6,8	1	47.96
	7,7	12	37.76
	8,4	7	36.73
100-norm	2,8	15	45.92
	3,8	8	50
	4,5	23	32.65
	5,4	23	32.65
	6,8	1	48.98
	7,7	12	37.76
	8,4	6	38.78

c) NN

For the nearest neighbor method, we wanted to find the optimal percentage of the total data range to look for neighbors and the best feature vectors. We also wanted to analyze this for the same distance measures used in the last two techniques. The table below (Table 5) shows the results, determined in MATLAB using the code listed in "nearn.m," attached at the end of this report. The nearest neighbor method seems to perform best with features 4, 5, and 7 and window sizes near 20% of the full data range.

Table 5: Nearest Neighbor Results

% of Range	1D		2D Manhattan		2D Euclidean		2D 100-norm	
	vector	% err	vector	% err	vector	% err	vector	% err
5	4	42.86	4, 7	42.86	4, 8	41.84	4, 4	42.86
10	7	41.84	4, 7	41.84	4, 5	36.73	4, 5	37.76
15	7	44.9	4, 7	44.9	4, 5	35.71	4, 5	36.73
20	4	41.84	4, 7	40.82	4, 7	40.82	4, 5	34.69
25	4	47.96	4, 7	43.88	4, 5	41.84	4, 5	40.82
30	4	46.94	4, 7	44.9	4, 5	45.92	4, 5	42.86

d) Comparison

Error percentages for all of these methods were consistently between 35% and 50% or so. This may seem rather high, and it is compared to the methods in question 2, but consider that there are 6 possible choices to make. Simply guessing will yield an 83.33% error rate on average (5/6). In all of the methods, using a higher norm, in this case 100, yielded better or at least as optimal values as the other distance measures and both norm measures did better than the Manhattan distance. The linear distance varied

too much to be sure how it matched up and since it always chose the best feature vector, its results are a bit different than the others who chose a pair.

Looking at the results for vectors 4, 5, and 7, K-Nearest Neighbors and the Parzen window methods generally beat out the Nearest Neighbor method but were themselves too close to say which did better with much conviction. Because, of the two, KNN is much easier to implement, it would be my choice of these three based on the data we obtained.

```
%ECE662HW2Q1.m
% %Data
% Africa = [20.74 70.99; 48.6  41; 42.2  54.41; 26.04 46.63; 45.93 50.65; 44.17 47.36;
%     37.87 49.86; 30.89 70.2; 37.92 43.31; 47.43 50.53; 36.53 62.97; 37.16 52.97;
%     37.53 46.83; 49.57 44.97; 31.4  53.37; 25.51 69.81; 39.77 49.35; 40.47 55.24;
%     40.69 50.73; 27.67 56.76; 38.09 58.04; 32.25 58.45; 41.99 53.68; 49.88 45.5;
%     39.11 50.99; 31.27 44.59; 49.87 43.75];
%
% Asia = [49.75 42.13; 11.19 71.4; 14.35 66.84; 19.3  74.8; 27.8  62.01; 22.37 63.46;
%     23.6  76.32; 27.49 56.75; 13.6  72.03; 8.08  81.5; 7.38  79.98; 12.14 78.96;
%     15.07 66.66; 11.13 70.48; 25.14 62.88; 20.67 68.59; 18.96 69.45; 35.59 57.05;
%     21.07 79.7; 8.95  81.86; 27.93 71.26; 16.67 64.88; 18.56 76.9; 20.98 65.35;
%     28.42 61.9];
%
% Europe = [17.24 75.65; 9.47  78.94; 9.27  68.41; 10.83 78.18; 9.4 74.09; 20.62 71.01;
%     8.74  72.36; 10.48 78.33; 9.1 74.88; 8.95  75.4; 12.01 77.31; 9.79  70.93;
%     10.95 78.37; 12.77 79.6; 8.67  78.73; 9.38  78.26; 9.46  72.43; 14.43 81.02;
%     15.17 77.78; 9.35  79.93 ];
%
% Africa_Test = [24.03 72.74; 39.26 57.29; 43.79 45.01; 48.59 51.82; 35.28 62.25;
%     15.89 71.95; 20.91 69.62; 43.53 44.03; 27.41 51.48; 51.16 54.5; 42.67 46.61;
%     19.94 75.74; 41.74 43.41; 34.87 64.34; 37.6  61.6; 46.93 41.01; 45.83 45.94;
%     24.07 53.37; 34.42 56.37; 30.42 43.87; 39.64 57.56; 17.07 73.04; 47.27 47.81;
%     42.12 49.65; 25.08 63.88; 41.92 39.19; 28.9  39.99];
%
% Asia_Test = [19.3  71.03; 22.74 73.04; 22.17 65.58; 19.65 65.03; 19.54 59.91;
%     30.25 61.34; 39.06 72.37; 23.54 74.19; 27.46 63.61; 28.08 70.3; 17.81 74.26;
%     10.39 77.05; 26.51 71.63; 10.13 78.77; 16.31 70.82; 28.23 73.05; 29.36 65.88;
%     15.37 68.56; 41.71 58.27; 19.48 70.85; 22.89 62.38; 16.66 77.81; 23.67 66.48;
%     20.15 73.02; 39.3  60.31];
%
% Europe_Test = [8.73  71.35; 9.05  72.05; 12.03 78.16; 10.04 78.62; 11.42 67.92;
%     13.2  73.98; 12.35 78.68; 12.35 79.28; 9.36  74.63; 10.89 77.24; 10.01 71.32;
%     9.89  64.79; 12.46 73.21; 9.68  73.81; 8.87  76.78; 10.24 79.99; 10.87 80.09;
%     9.8 80.69; 11.99 73.44; 8.45  67.59];
%
% mean_1 = -8;
% mean_2 = 8;
% variance = 10;
%
% %Plant data
% for i = 1:length(Africa)
%     Africa(i, 1:2) = [normrnd(mean_1, variance), normrnd(mean_1, variance)];
%     Africa_Test(i, 1:2) = [normrnd(mean_1, variance), normrnd(mean_1, variance)];
% end
% for i = 1:length(Europe)
%     Europe(i, 1:2) = [normrnd(mean_2, variance), normrnd(mean_2, variance)];
%     Europe_Test(i, 1:2) = [normrnd(mean_2, variance), normrnd(mean_2, variance)];
% end

%Find the means of two data sets (in this case, Africa & Europe)
m_Africa = [0 0];

for i = 1:length(Africa)
  m_Africa(1) = m_Africa(1) + Africa(i, 1);
  m_Africa(2) = m_Africa(2) + Africa(i, 2);
end

m_Africa(1) = m_Africa(1) / length(Africa);
m_Africa(2) = m_Africa(2) / length(Africa);
```

```
m_Europe  = [0 0];

for i = 1:length(Europe)
  m_Europe(1) = m_Europe(1) + Europe(i, 1);
  m_Europe(2) = m_Europe(2) + Europe(i, 2);
end

m_Europe(1) = m_Europe(1) / length(Europe);
m_Europe(2) = m_Europe(2) / length(Europe);

%Calculate the between class scatter
S_B = (m_Africa' - m_Europe') * (m_Africa - m_Europe);

%Calculate the within class scatter
S_W_Africa = [0 0; 0 0];
for i = 1:length(Africa)
  S_W_Africa = S_W_Africa + ((Africa(1, 1:2)' - m_Africa') * (Africa(1, 1:2) - m_Africa));
end

S_W_Europe = [0 0; 0 0];
for i = 1:length(Europe)
  S_W_Europe = S_W_Europe + ((Africa(1, 1:2)' - m_Europe') * (Europe(1, 1:2) - m_Europe));
end

S_W = S_W_Africa + S_W_Europe;

%Create line to plot through the data (y & x)
x = -400:300;
t = ((m_Africa' + m_Europe') / 2.0);
%The optimal w value, as calculated in class
%w = inv(S_W) * (m_Africa' - m_Europe');
w = [-1 0]; %The commented w's are the maximized value without considering the within
%class scatter (switch with the uncommented w in order to change the results)
z = m_Africa' - m_Europe';
w(2) = z(1) / z(2);
y = (x ./ (w(1) .* t(1))) .* (w(2) .* t(2));

%Test Data
errors_Africa = 0;
for i = 1:length(Africa)
  if Africa_Test(i, 2) > (Africa_Test(i, 1) / (w(1) * t(1))) * (w(2) .* t(2))
    errors_Africa = errors_Africa + 1;
  end
end

%Calculate and display errors in classifying the first class
errors_Africa / length(Africa_Test)

errors_Europe = 0;
for i = 1:length(Europe)
  if Europe_Test(i, 2) < (Europe_Test(i, 1) / (w(1) * t(1))) * (w(2) .* t(2))
    errors_Europe = errors_Europe + 1;
  end
end

%Calculate and display errors in classifying the second class
errors_Europe / length(Europe_Test)

%Plot data
```

```
hold on
for i = 1:length(Africa_Test)
  plot(Africa_Test(i, 1), Africa_Test(i, 2), 'r*')
end

for i = 1:length(Europe_Test)
  plot(Europe_Test(i, 1), Europe_Test(i, 2), 'b*')
end

plot(x, y)
hold off
```

```
//xor_train.c
/*
Fast Artificial Neural Network Library (fann)
Copyright (C) 2003 Steffen Nissen (lukesky@diku.dk)

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
*/

#include <stdio.h>

#include "fann.h"
#include "floatfann.h"

int FANN_API test_callback(struct fann *ann, struct fann_train_data *train,
    unsigned int max_epochs, unsigned int epochs_between_reports,
    float desired_error, unsigned int epochs)
{
    printf("Epochs     %8d. MSE: %.5f. Desired-MSE: %.5f\n", epochs, fann_get_MSE(ann), desired_
    return 0;
}

int main()
{
    FILE *opfile = fopen("clo.txt", "w");
    fann_type *calc_out;
    //The input parameters
    const unsigned int num_input = 7;
    const unsigned int num_output = 6;
    const unsigned int num_layers = 3;
    const unsigned int num_neurons_hidden = 3;
    const float desired_error = (const float) 0.01;
    const unsigned int max_epochs = 1000;
    const unsigned int epochs_between_reports = 10;
    struct fann *ann;
    struct fann_train_data *data;
    struct fann_train_data *test_data;
    float mse; //Mean squared error

    unsigned int i = 0;
    unsigned int decimal_point;

    fprintf(opfile, "Creating network.\n");
    ann = fann_create_standard(num_layers, num_input, num_neurons_hidden, num_output);

    data = fann_read_train_from_file("TrainingDataNN.data");

    fann_set_activation_steepness_hidden(ann, 1);
    fann_set_activation_steepness_output(ann, 1);
```

```
    //Functions available(we varied the hidden function)
    fann_set_activation_function_hidden(ann, FANN_SIGMOID_SYMMETRIC);
    fann_set_activation_function_output(ann, FANN_SIGMOID_SYMMETRIC);

    fann_set_train_stop_function(ann, FANN_STOPFUNC_BIT);
    fann_set_bit_fail_limit(ann, 0.01f);

    fann_init_weights(ann, data);

    fprintf(opfile, "Training network.\n");
    fann_train_on_data(ann, data, max_epochs, epochs_between_reports, desired_error);

    for(i = 0; i < fann_length_train_data(data); i++)
    {
        calc_out = fann_run(ann, data->input[i]);
        fprintf(opfile, "Test (%f,%f) -> %f, should be %f, difference=%f\n",
               data->input[i][0], data->input[i][1], calc_out[0], data->output[i][0],
               fann_abs(calc_out[0] - data->output[i][0]));
    }

    fprintf(opfile, "Saving network.\n");

    fann_save(ann, "world_fixed_float.net");

    decimal_point = fann_save_to_fixed(ann, "world_fixed.net");
    fann_save_train_to_fixed(data, "world_fixed.data", decimal_point);

    fprintf(opfile, "Cleaning up.\n");
    fann_destroy_train(data);

    //Test
    test_data = fann_read_train_from_file("TestDataNN.data");
    mse = fann_test_data(ann, test_data);
    fprintf(opfile, "MSE for test data: %f", mse);

    fann_destroy(ann);
    fann_destroy_train(test_data);
    fclose(opfile);
    return 0;
}
```

```
#svm.py
import sys
import PyML
from PyML import *

fileName = "trained.tf"

def Train():
    #load training data file
    dataTrain = datafunc.SparseDataSet('TrainingDataSVM.data')
    #create and save SVM, trained against the data
    s = svm.SVM()
    s.train(dataTrain, saveSpace = False)
    s.save(fileName)

def Test():
    #load test data file
    dataTest = datafunc.SparseDataSet('TestDataSVM.data')
    #load already-prepared SVM
    loadedSVM = svm.loadSVM(fileName)
    #test svm and output results
    r = loadedSVM.test(dataTest)
    print r.getDecisionFunction()
    print r.getSuccessRate()
    print r.getInfo()
    r.plotROC()
    print r.getLog()

#First, train the SVM on the training data
Train()
#Next, test the SVM against the test data
Test()
```

```
%parzwin.m
% Parzen Window

dist = 0;
trn = 0;
tst = 0;
prcterr = 0;

% create training and test matrices with know classes in first column and
% feature vectors in column 2-8
trn = createtrn();
tst = createtst();

% set variance of Gaussian window to use later
var = 4;

for m=2:8;
    for n=2:8
        error = 0;
        % Go through each test point
        for j=1:98
            % Go through each training point
            for i=1:97
                % assign class of training point to link to the distance
                dist(i,1) = tst(i,1);
            % Calculate distance of training points from test point
                %dist(i,2) = abs((tst(j,n) - trn(i,n)));    % 1D
                %dist(i,2) = abs((tst(j,m) - trn(i,m)) + (tst(j,n) - trn(i,n)));         %2D man
                %dist(i,2) = ((tst(j,m) - trn(i,m))^2 + (tst(j,n) - trn(i,n))^2)^(1/2);   %2D eu
                %dist(i,2) = ((tst(j,m) - trn(i,m))^100 + (tst(j,n) - trn(i,n))^100)^(1/100);
            end

            %Sort rows in decending order (closest first)
            dist = sortrows(dist,2);

            % apply Gaussian parzen window with coordinate system
            % translated to have the test point located at the origin
            % and calculate weight of training point
            for i=1:97
                dist(i,3) = normpdf(0,dist(i,2),var);
            end

            cntnt = [0 0 0 0 0 0];

            % count weighted votes for all training points
            for i=1:97
                cntnt(dist(i,1))=cntnt(dist(i,1))+dist(i,3);
            end

            %line votes up with respective classes
            cntclss = [1 cntnt(1); 2 cntnt(2); 3 cntnt(3); 4 cntnt(4); 5 cntnt(5); 6 cntnt(6)];

            % sort classes based on election (lowest first)
            % assign last row's class (highest votes) to test vector
            cntclss = sortrows(cntclss,2);
            tst(j,9) = cntclss(6,1);

            % compare test vector's assigned class with actual class and
            % increase error appropriately
            if (tst(j,1) ~= tst(j,9))
```

```
                error = error + 1;
            end
        end
        % calculate percent error for each iteration
        prcterr(var,n-1) = error/98 * 100;
    end
end
```

```
%knearn.m
% K Nearest Neighbor

dist = 0;
trn = 0;
tst = 0;
prcterr = 0;

% create training and test matrices with know classes in first column and
% feature vectors in column 2-8
trn = createtrn();
tst = createtst();

% set K number of points to use later
Krad = 6;

%compare each test vector to each training vector
for m = 2:8
    for n=2:8
        error = 0;
        % Go through each test point
        for j=1:98
            % Go through each training point
            for i=1:97
                % assign class of training point to link to the distance
                dist(i,1) = tst(i,1);
            % Calculate distance of training points from test point
                %dist(i,2) = abs((tst(j,n) - trn(i,n)));    % 1D
                %dist(i,2) = abs((tst(j,m) - trn(i,m)) + (tst(j,n) - trn(i,n)));         %2D man
                %dist(i,2) = ((tst(j,m) - trn(i,m))^2 + (tst(j,n) - trn(i,n))^2)^(1/2);   %2D eu
                %dist(i,2) = ((tst(j,m) - trn(i,m))^100 + (tst(j,n) - trn(i,n))^100)^(1/100);
            end

            %Sort rows in decending order (closest first)
            dist = sortrows(dist,2);

            cntnt = [0 0 0 0 0 0];

            % count first Krad votes
            for k=1:Krad
                cntnt(dist(k))=cntnt(dist(k))+1;
            end

            %line votes up with respective classes
            cntclss = [1 cntnt(1); 2 cntnt(2); 3 cntnt(3); 4 cntnt(4); 5 cntnt(5); 6 cntnt(6)];

            % sort classes based on election (lowest first)
            % assign last row's class (highest votes) to test vector
            cntclss = sortrows(cntclss,2);
            tst(j,9) = cntclss(6,1);

            % compare test vector's assigned class with actual class and
            % increase error appropriately
            if (tst(j,1) ~= tst(j,9))
                error = error + 1;
            end
        end
        % calculate percent error for each iteration
        prcterr(Krad,n-1) = error/98 * 100;
    end
```

```
end
```

```
%nearn.m
% Nearest Neighbor

dist = 0;
trn = 0;
tst = 0;
prcterr = 0;

% create training and test matrices with know classes in first column and
% feature vectors in column 2-8
trn = createtrn();
tst = createtst();

% set precentage of range to be used later
perc = .3;

%compare each test vector to each training vector
for m=2:8
    for n=2:8
        error = 0;
        % Go through each test point
        for j=1:98
            % Go through each training point
            for i=1:97
                % assign class of training point to link to the distance
                dist(i,1) = tst(i,1);
            % Calculate distance of training points from test point
                %dist(i,2) = abs((tst(j,n) - trn(i,n)));            %1D
                %dist(i,2) = abs((tst(j,m) - trn(i,m)) + (tst(j,n) - trn(i,n)));         %2D man
                %dist(i,2) = ((tst(j,m) - trn(i,m))^2 + (tst(j,n) - trn(i,n))^2)^(1/2);   %2D eu
                %dist(i,2) = ((tst(j,m) - trn(i,m))^100 + (tst(j,n) - trn(i,n))^100)^(1/100);
            end

            %Sort rows in decending order (closest first)
            dist = sortrows(dist,2);

            %determine range of distance vector
            valrange = dist(97,2)-dist(1,2);
            %determine an effective stopping point based on values of
            %distance
            Vrad = dist(1,2) + perc * valrange;

            % reset values
            cntnt = [0 0 0 0 0 0];
            k = 1;

            % count the votes for each class up to pre-determined stopping
            % point
            while (dist(k,2) < Vrad)
                cntnt(dist(k))=cntnt(dist(k))+1;
            k = k+1;
            end

            %line votes up with respective classes
            cntclss = [1 cntnt(1); 2 cntnt(2); 3 cntnt(3); 4 cntnt(4); 5 cntnt(5); 6 cntnt(6)];

            % sort classes based on election (lowest first)
            % assign last row's class (highest votes) to test vector
            cntclss = sortrows(cntclss,2);
            tst(j,9) = cntclss(6,1);
```

```
            % compare test vector's assigned class with actual class and
            % increase error appropriately
            if (tst(j,1) ~= tst(j,9))
                error = error + 1;
            end
        end
        % calculate percent error for each iteration
        prcterr(m-1,n-1) = error/98 * 100;
    end
end
```