Evolutionary Programming/Evolutionary Strategy vs. Grid Search Training Method

Hemant Heer, Josh Cohn, Kyle Bird
BE 340 Intro to Bioinformatics
Professor Walker Land

Contents

Background Information

When cancer metastasizes it can spread to other parts of the body. Cancer cells of many types can gain entry into the pleural cavity, or the space that surrounds the lungs. Many times the presence of cancer cells in the pleural cavity will cause an accumulation of fluid, known as pleural effusion. Excess fluid is due to cancer cells plugging the microscopic drainage sites in the pleural lining where the body can normally absorb fluid. It is also due to the direct secretion of fluid by the cancer cells themselves. Extra fluid in the pleural cavity displaces the lungs, which can lead to shortness of breath and overall chest discomfort. Over 50% of people who have cancer will develop pleural effusion. One way of diagnosing pleural effusion due to the presence of cancer cells is through thoracentisis. Thoracentisis is the process of extracting fluid of the pleural effusion with a needle and then examining it under a microscope in order to visually classify whether the cells are benign or malignant. For some cells it is easy to visually classify whether benign or malignant. What about for atypical cells? We need a second support system to help differentiate between the two classes based on non-visual clues.

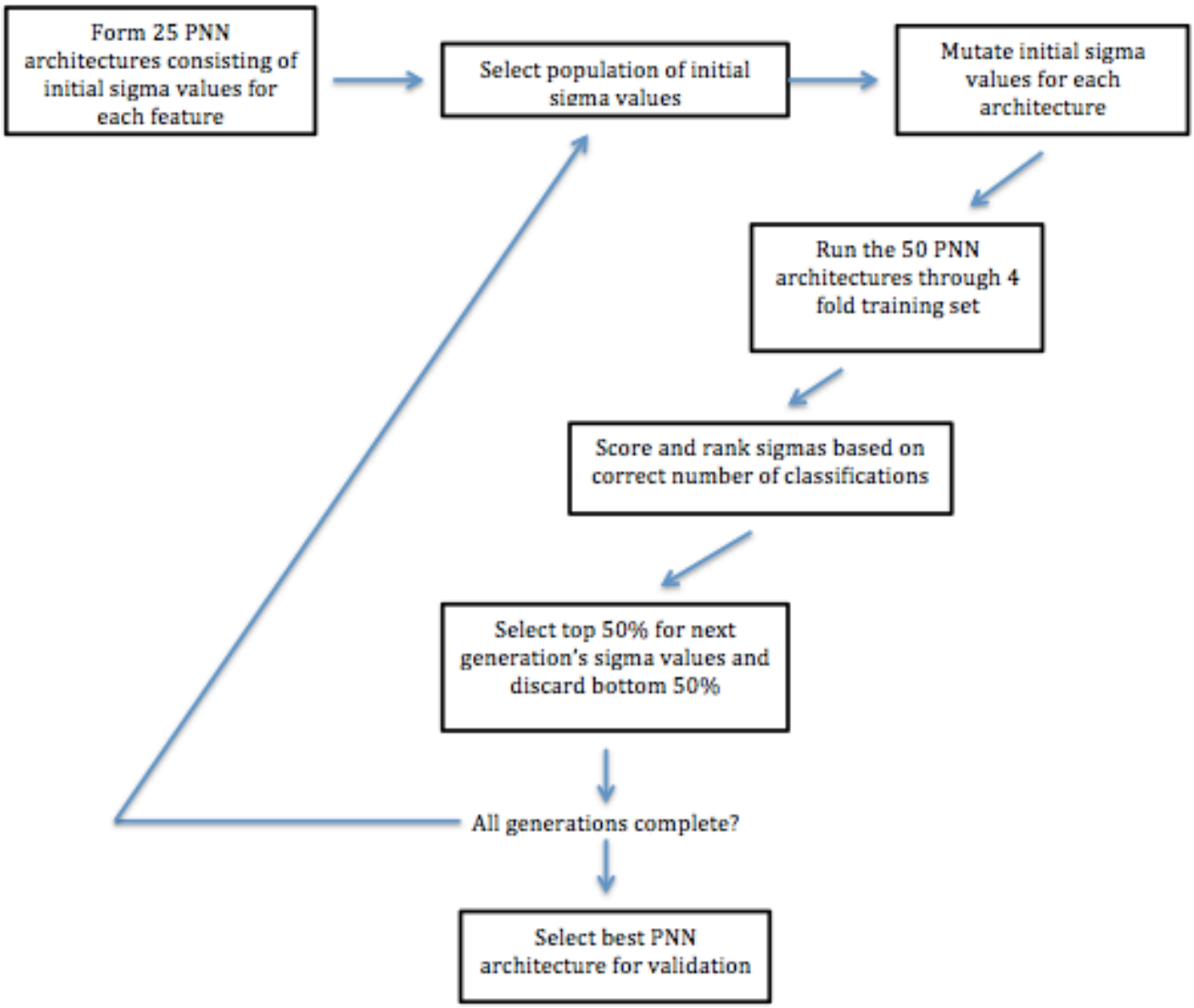
Problem Addressed

For our project we were given 518 samples from patients where the pathologist was able to classify whether benign or malignant based off of visual clues. Two hundred and eighty three of the samples were malignant and 235 of the samples were benign. The specific problem we addressed within our project was to compare the training of a PNN using both the standard grid search method and the evolutionary programming/evolutionary strategy method. One of the more important aspects of a PNN is its training process. Training occurs through manipulation of the sigma parameters in the pattern layer. Goal is to determine the best value of sigma, the smoothing parameter. We trained our PNN using these two methods and then validated using a 5-fold cross validation. For the grid search method we look at each sigma of each feature individually. We bound the sigma value to some arbitrary interval and then test the performance of each "test" sigma at equal increments. We then evaluate each sample at an initial increment and then increase that sigma by a given increment. For the EP/ES method we look at the features of a sample as a population of sigmas. We evolve the sigmas for a population of PNN models. Population of sigmas is randomly generated (one for each feature). Each of the sigmas is then copied and mutated, which yields a set of parameters twice the original size. We then run the sigmas through the training set and rank based on performance. Upper 50% of solution pool is selected as the basis for the next generation pool and the bottom 50% is killed off so the sigma pool reduces to the original size. Process is then repeated for an arbitrary number of generations.

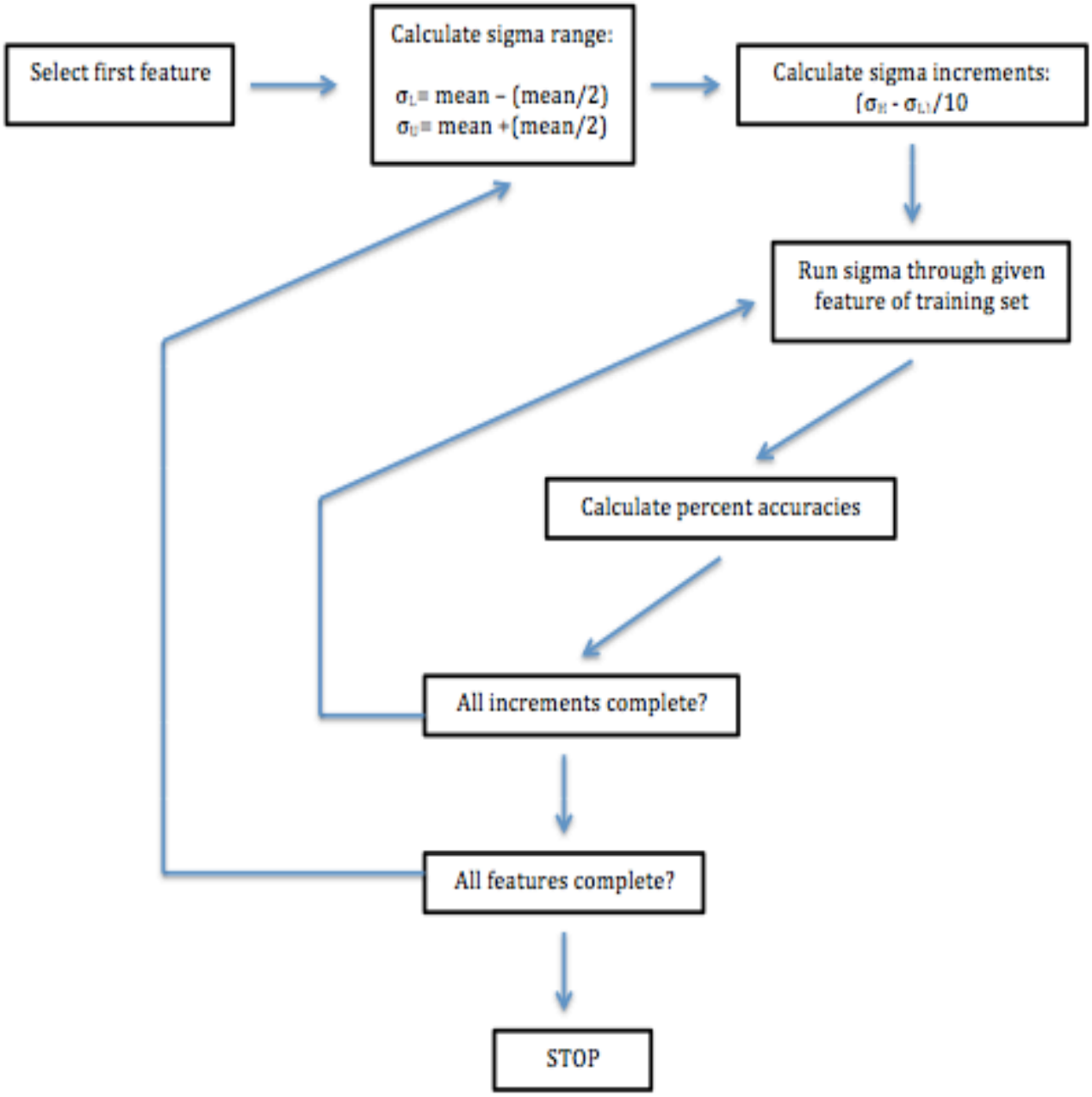
HIPO

Input	Process	Output
518 patient data collections from pleural effusion collection. Data is in the form of five folds	Form 5 training sets by combining the folds, excluding a different fold each time	5 separate training data sets to train the PNN
Training sets: Folds 1,2,3,4,5 while excluding a single fold each time	Generate initial sigma values for each feature and PNN architecture for first population between 0.005 and 0.0055	Population of 300 sigma values (12 features x 25 PNN architectures)
Population consisting of 25 sets of initial sigma values	Mutate sigmas: $\sigma_i' = \sigma_i \exp\left\{\frac{1}{\sqrt{2n}}N(0,1) + \frac{1}{\sqrt{2\sqrt{n}}}N_i(0,1)\right\}$	
	Update mutation vector $x_i' = x_i + C\sigma' i$	50 sigma sets (25 initial sigma values and 25 mutated sigma values)
All sigma sets of given population (now double in size)	Calculate the pattern layer function: $D(x,x_i) = (x,x_i)^2/2\sigma^2$	Pattern layer function $D(x,x_i)$ for each feature
$D(x,x_i)$	$f_{ai}(x) = 1/(2\pi)^{1/2}\sigma(1/n)\Sigma Exp[-D(x,x_i)]$	$f_{ai}(x)$& $f_{bi}(x)$ functions for each feature for each sigma
$f_{ai}(x)$& $f_{bi}(x)$ functions	Compute the decision rule: $f_{ai}(x) > f_{bi}(x)$....0 $f_{ai}(x) < f_{bi}(x)$....1	Output of the decisions made for each sample
Decision Output	Compare against the diagnosis from visual classification ("gold standard") and compute the accuracy for the sets sigma values	50 accuracies per generation for the given population
50 total accuracies (1 from each sigma set)	Rank each sigma set based on correct number of classifications. Select top 50% and discard bottom 50%	Top 25 sets of sigma values for the given population
Best performing sigma set and the excluded fold	Validate by running the fold through the PNN using the same process but only for the selected sigma set	5 accuracies from each fold that can be used to calculate the overall accuracy

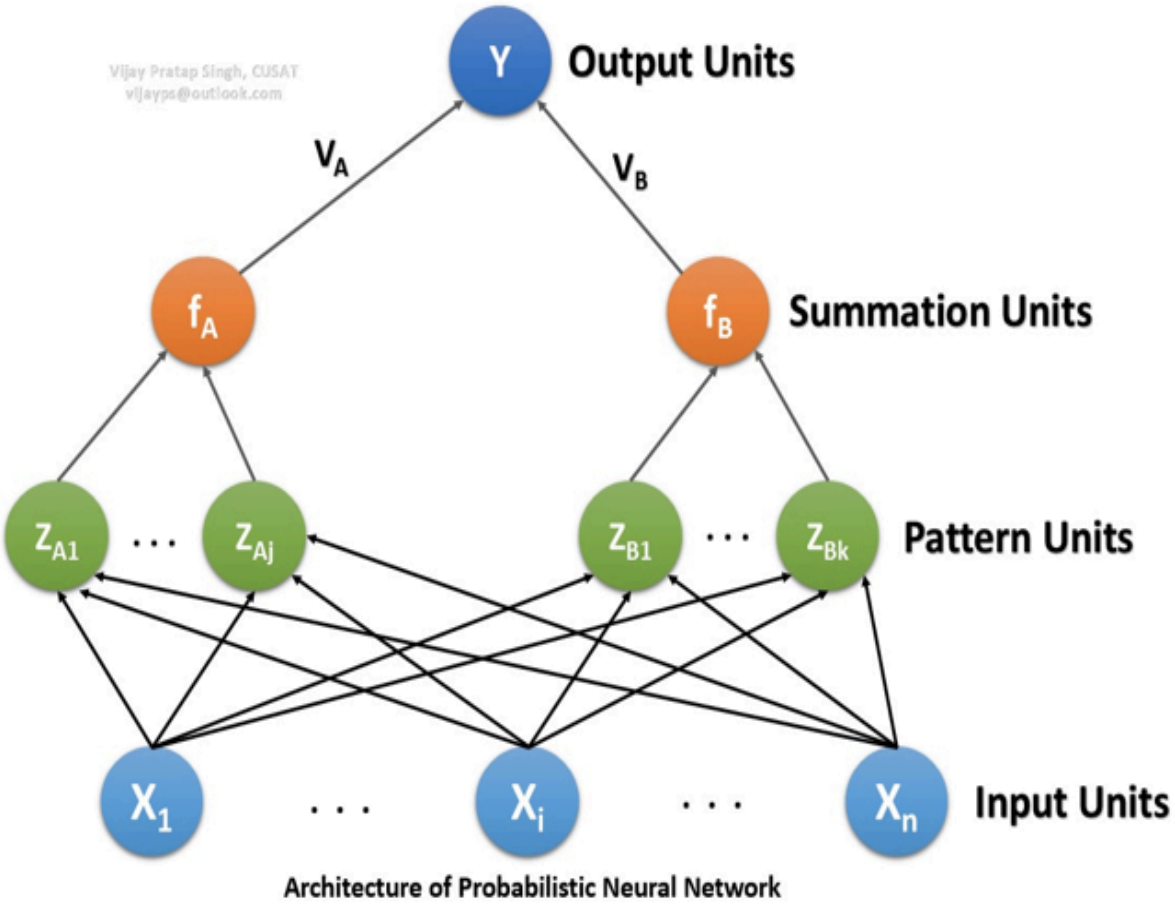
EP/ES Basic Training Flow Chart



"Grid Search" Basic Training Flow Chart



PNN Architecture



Architecture of Probabilistic Neural Network

For our project we had 518 samples, 12 features, and 2 decision classes. The architecture of our training and validation consisted of a five-fold cross validation:

12 Input layer neurons (12 features)
412 Pattern layer neurons (103 samples in each fold so 412 samples for 4 folds)
2 Summation layer neurons (2 decision classes: benign/malignant)
1 Output neuron

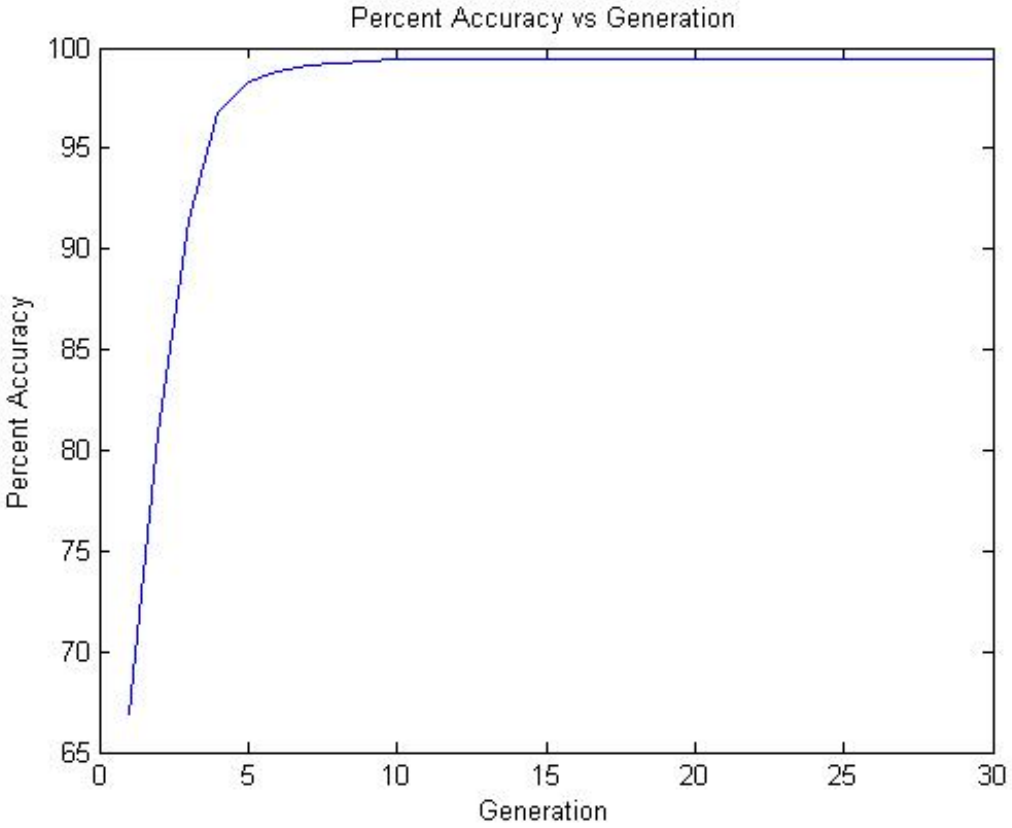
Five-Fold Cross Validation

1. Separate data into Folds (5 for this project)
2. Train on 4/5 of the data
3. Validate that fold
4. Repeat until all folds have been validated

This five fold cross validation was incorporated into EP/ES because all the percent accuracies after validating each fold for all the best sigmas were averaged.

As for results, since the data was separated into folds randomly, the accuracies varied from 95-99% using grid search and the accuracies started from 50% and increased to 99% as the generations increased using the EP/ES technique.

Results



Increasing percent accuracy curve with number of generations and approaches a maximum accuracy threshold after a sufficient number of generations. Starting from 50% and increases to 99% by the 30th generation and if this is run for many number of times the accuracy tends to approach 100%, however such a classification system does not exist as it is incredibly unlikely to have a perfect system. Although the numbers aren't displayed below(primarily because the sigma values are randomly generated every generation) the population of sigmas have higher sigma values because they are added to the previous generation of sigmas (the initial population).

Conclusions

Based our on results we concluded that the Evolutionary Programming, Evolutionary Strategy training method is a better training method than the standard "grid search" method. While both training methods resulted in high classification accuracy, the EP/ES training method was able to reach the optimal sigma value in a quicker time period. Part of the reason why we believe the EP/ES method is much better at training a PNN is because the grid search method simply sets an arbitrary bound on the sigma values and then tests each sigma value at equal increments. If a sigma value performs poorly, it does not affect the next iteration in training. An arbitrary increment is added to that sigma value and the process is repeated. Thus, it is simply "searching" for the best sigma value rather than actually working its way training towards the best sigma value. The EP/ES training method however employs a mutation and selection process, making it more efficient. After each iteration only the best sigma values are being passed on for the next generation. That means that after each iteration the training accuracy will get better and better, unlike the grid search method where the accuracies can oscillate back and forth. Secondly, the grid search method looks at each individual feature as a separate part of the system and does not count for any connections that might exist between the given features. We know that in biology features are not independent, but interact and are influenced by one another. For this reason training each feature individually in not a good representative of how the features actually impact the decision class in real life. However, the EP/ES training method looks at the features collectively when training the PNN model and is a better indicator of what actually occurs in the real world.

Recommendations

Based on our results and conclusions from those results our group would recommend using the Evolutionary Programming and Evolutionary Strategy training method when designing and training a PNN system. Because training a PNN is such a crucial part of the decision process, it is important to employ the best method possible that optimizes the best sigma value the quickest. It is also our recommendation for the future to test our training method on a harder to classify set of data. The data our group received was 518 samples from patients where the pathologist had already been able to visually classify whether the sample was benign or malignant. Also, the original data set consisted of over 200 features but the data we used consisted of only 12 features that were hand picked because of the likelihood that they would provide the best results. Unfortunately, the results were almost too good, making it harder to determine if one training method really was better than the other. We believe that given a harder to classify data set there will be a greater separation in the ability to classify between the grid search method and the EP/ES training method.

User's Manual

The software package used to write the code for the probabilistic neural network was Matlab which comes with all the necessary in built mathematical functions. Matlab functions around the use of arrays and vectors are a necessary part of all programs that are written. In order to use the user's manual I would suggest creating a new directory in your operating system of choice (Windows, Linux, or Mac OS X) and putting all the necessary scripts and functions into that directory. Matlab can run programs if they are not in the folder that you have set in the software package but that requires adding the path to Matlab when running the script. For the sake of simplicity I would create a new folder on your computer and put these files in: create_initial_population, EPES_Accuracy_Sigma_Mutated_Sigmas, makeControlsCases,mutate_sigmas, shuffleData,PNN_Training,PNN_Validation, NORMALIZED_BENIGN_ForJuniors,NORMALIZED_MALIGNANT_ForJuniors.

After these files are placed in a directory of your choice, open Matlab and above the command window there will be a tool bar to select directories or manually enter the directory that contains the necessary files. Set the current directory in Matlab to that particular folder. The only file that you have to run is EPES_Accuracy_Sigma_Mutated_sigmas. Once this script file has been opened, pres F5 to run the entire program and several variables will have shown up in the workspace window of Matlab's window. In addition, a graph of percent accuracy as a function of generation number will open and show the accuracy of the classification over time.

The previous information was applicable for EP/ES information and this information will be concerned with the traditional grid search method. Feel free to create a new directory and place these specified files: NORMALIZED_BENIGN_ForJuniors, NORMALIZED_MALIGNANT_ForJuniors, shuffleData, makeControlsCases,makeSigs, PNN_Validation, pnnTrain, GRID_SEARCH_PNN_RESULTS_ALL. In order to run the grid search method you only need to open GRID_SEARCH_PNN_RESULTS_ALL and press F5 to calculate a total accuracy for the 5 fold cross validation.

Appendices/Code

Grid Search method scripts:

ALL_IN_ONE FILE

```
% Hemant Heer
% BE 340
% 12/10/2013

clc;clear all;


orig_controls = csvread('NORMALIZED_BENIGN_ForJuniors.csv',1);
orig_cases = csvread('NORMALIZED_MALIGNANT_ForJuniors.csv',1);
complete = [orig_controls;orig_cases];

[f1,f2,f3,f4,f5] = shuffleData(complete);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dat = [f2;f3;f4;f5]; %Fold 1 removed

dataDim = size(dat); %Dimensions of imported data

[controls,cases] = makeControlsCases(dat);


[siglows,sighighs] = makeSigs(dat);

n = 100;
deltasig = (max(sighighs) - min(siglows))./n
sigmas = min(siglows):deltasig:max(sighighs);
acc = pnnTrain(n,sigmas,dataDim,controls,cases);

for j = 1:size(acc,2)

   maxAcc = max(acc(:,j));
   maxIndex = find(acc(:,j)==maxAcc,1);
   fprintf('The first index at which the maximum percent accuracy appears is %d
\n',maxIndex);
   fprintf('This corresponds to a sigma value of %f \n',sigmas(1,j));
   best_sigmas(1,j) = sigmas(1,j);
```

```
end

p1= PNN_Validation(f1,dat,best_sigmas,controls,cases);

%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%
dat = [f1;f3;f4;f5]; %Fold 2 removed

dataDim = size(dat); %Dimensions of imported data

[controls,cases] = makeControlsCases(dat)

[siglows,sighighs] = makeSigs(dat);


n = 100;
deltasig = (max(sighighs) - min(siglows))./n
sigmas = min(siglows):deltasig:max(sighighs);
acc = pnnTrain(n,sigmas,dataDim,controls,cases);

for j = 1:size(acc,2)

   maxAcc = max(acc(:,j));
   maxIndex = find(acc(:,j)==maxAcc,1);
   fprintf('The first index at which the maximum percent accuracy appears is %d
\n',maxIndex);
   fprintf('This corresponds to a sigma value of %f \n',sigmas(1,j));
   best_sigmas(1,j) = sigmas(1,j);

end

p2= PNN_Validation(f2,dat,best_sigmas,controls,cases);

%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%
dat = [f1;f2;f4;f5]; %Fold 3 removed

dataDim = size(dat); %Dimensions of imported data

[controls,cases] = makeControlsCases(dat)

[siglows,sighighs] = makeSigs(dat);

n = 100;
```

```
deltasig = (max(sighighs) - min(siglows))./n
sigmas = min(siglows):deltasig:max(sighighs);
acc = pnnTrain(n,sigmas,dataDim,controls,cases);

for j = 1:size(acc,2)

   maxAcc = max(acc(:,j));
   maxIndex = find(acc(:,j)==maxAcc,1);
   fprintf('The first index at which the maximum percent accuracy appears is %d
\n',maxIndex);
   fprintf('This corresponds to a sigma value of %f \n',sigmas(1,j));
   best_sigmas(1,j) = sigmas(1,j);

end

p3 = PNN_Validation(f3,dat,best_sigmas,controls,cases);

%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dat = [f1;f2;f3;f5]; %Fold 4 removed

dataDim = size(dat); %Dimensions of imported data

[controls,cases] = makeControlsCases(dat);

[siglows,sighighs] = makeSigs(dat);

n = 100;
deltasig = (max(sighighs) - min(siglows))./n
sigmas = min(siglows):deltasig:max(sighighs);
acc = pnnTrain(n,sigmas,dataDim,controls,cases);

for j = 1:size(acc,2)

   maxAcc = max(acc(:,j));
   maxIndex = find(acc(:,j)==maxAcc,1);
   fprintf('The first index at which the maximum percent accuracy appears is %d
\n',maxIndex);
   fprintf('This corresponds to a sigma value of %f \n',sigmas(1,j));
   best_sigmas(1,j) = sigmas(1,j);

end

p4 = PNN_Validation(f4,dat,best_sigmas,controls,cases);
```

```
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dat = [f1;f2;f3;f4]; %Fold 5 removed

dataDim = size(dat); %Dimensions of imported data

[controls,cases] = makeControlsCases(dat);

[siglows,sighighs] = makeSigs(dat);

n = 100;
deltasig = (max(sighighs) - min(siglows))./n
sigmas = min(siglows):deltasig:max(sighighs);
acc = pnnTrain(n,sigmas,dataDim,controls,cases);

for j = 1:size(acc,2)

   maxAcc = max(acc(:,j));
   maxIndex = find(acc(:,j)==maxAcc,1);
   fprintf('The first index at which the maximum percent accuracy appears is %d
\n',maxIndex);
   fprintf('This corresponds to a sigma value of %f \n',sigmas(1,j));
   best_sigmas(1,j) = sigmas(1,j);

end

p5 = PNN_Validation(f5,dat,best_sigmas,controls,cases);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TOTAL_ACCURACY_5_FOLDS =
(mean(p1,2)+mean(p2,2)+mean(p3,2)+mean(p4,2)+mean(p5,2))./5
```

<u>shuffleData</u>

```
function [f1,f2,f3,f4,f5] = shuffleData(data_input)
% shuffleData: This function randomly shuffles the rows so 5 separate
% folds can be created in order to run a 5-fold validation



shuffledComplete = data_input(randperm(size(data_input,1)),:);

f1 = shuffledComplete(1:105,:);
f2 = shuffledComplete(106:211,:);
```

```
f3 = shuffledComplete(212:317,:);
f4 = shuffledComplete(318:423,:);
f5 = shuffledComplete(424:518,:);

end
```

makeSigs

```
function [siglows,sighighs] = makeSigs(dat)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
sig1low = mean(dat(:,1))-mean(dat(:,1))./2;
sig2low = mean(dat(:,2))-mean(dat(:,2))./2;
sig3low = mean(dat(:,3))-mean(dat(:,3))./2;
sig4low = mean(dat(:,4))-mean(dat(:,4))./2;
sig5low = mean(dat(:,5))-mean(dat(:,5))./2;
sig6low = mean(dat(:,6))-mean(dat(:,6))./2;
sig7low = mean(dat(:,7))-mean(dat(:,7))./2;
sig8low = mean(dat(:,8))-mean(dat(:,8))./2;
sig9low = mean(dat(:,9))-mean(dat(:,9))./2;
sig10low = mean(dat(:,10))-mean(dat(:,10))./2;
sig11low = mean(dat(:,11))-mean(dat(:,11))./2;
sig12low = mean(dat(:,12))-mean(dat(:,12))./2;


sig1high = mean(dat(:,1))+mean(dat(:,1))./2;
sig2high = mean(dat(:,2))+mean(dat(:,2))./2;
sig3high = mean(dat(:,3))+mean(dat(:,3))./2;
sig4high = mean(dat(:,4))+mean(dat(:,4))./2;
sig5high = mean(dat(:,5))+mean(dat(:,5))./2;
sig6high = mean(dat(:,6))+mean(dat(:,6))./2;
sig7high = mean(dat(:,7))+mean(dat(:,7))./2;
sig8high = mean(dat(:,8))+mean(dat(:,8))./2;
sig9high = mean(dat(:,9))+mean(dat(:,9))./2;
sig10high = mean(dat(:,10))+mean(dat(:,10))./2;
sig11high = mean(dat(:,11))+mean(dat(:,11))./2;
sig12high = mean(dat(:,12))+mean(dat(:,12))./2;

siglows =
[sig1low;sig2low;sig3low;sig4low;sig5low;sig6low;sig7low;sig8low;sig9low;sig10low;si
g11low;sig12low];
sighighs =
[sig1high;sig2high;sig3high;sig4high;sig5high;sig6high;sig7high;sig8high;sig9high;sig1
0high;sig11high;sig12high];

end
```

makeControlsCases

```
function[controls, cases] = makeControlsCases(dat)

controls = [];  %Define empty sets for controls and cases
cases = [];
%     cases = zeros(234,13);
%     controls = zeros(180,13);
for i = 1:size(dat,1)    %Scan each row of the normalized data
   if dat(i,size(dat,2))==0     %Unnormalized data has the same dimensions of entire data
      controls = [controls;dat(i,:)];
   else
      cases = [cases;dat(i,:)];
   end
end

end
```

PNN_Validation

```
function [percentAccuracy] = PNN_Validation(f1,dat,sigmas,controls,cases)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RUN THE NEURAL NETWORK

%Pre-allocation distCas and distCon
distCas = zeros(size(controls,1),size(f1,1));
distCon = zeros(size(controls,1),size(f1,1));
percentAccuracy = zeros(size(sigmas,1),1);

for i = 1:size(sigmas,1) %Iterate on sigma sets

   current_best_sigmas = sigmas(i,:); %Pick the set of sigmas on the row we are on

   %Set an initial variable to 1 so the sigma product can be calculated
   %Sort of works like an accumulator variable
   sigProd = 1;

   % For the current set of sigmas that we are on keep multiplying the
   % value in sigProd by the sigma value in the sigma set that was picked
   % out by the iterator i. j will iterate on all the values of sigma and
   % calculate a total product by mutiplying sigprod by the next value of
   % sigma in the set picked out.
   for j = 1:size(current_best_sigmas,2)
      sigProd = sigProd.*current_best_sigmas(1,j);
```

```
    end

    % Initialize an iterator k to 1 because k will iterate on the number
    % of controls in the while loop. Since we are in the pattern layer, a
    % sample in the pattern layer is picked a Euclidean distance is
    % calculated using this formula:
    % f1(i,j) - controls(k,j)).^2./(2.*sigmas(1,j).^2)
    % k will be incremented by 1 to move on to the next neuron(sample in
    % the controls class of the pattern layer)
    % Each sample from both classes in the pattern layer has 12 features
    % associated with it and we access it by the classname
    k = 1;

    while k <=size(controls,1)

        % We now iterate on the number of samples in fold 1(the one that
        % was removed, starting from the first sample all the way to the
        % end. i is simply accessing the sample we are picking. If we are
        % considering a matrix, then you can picture having l lines of
        % samples within the fold
        for l = 1:size(f1,1)

            % We now initiailize an accumulator variable to 0 because we
            % now have calculate the Euclidean Distance and sum of the
            % differences between each feature of the neuron in the pattern
            % layer and the feature in the sample being sent through the
            % controls class of the pattern layer. This value of the
            % Euclidean Distance is the difference between the ith sample
            % and the jth feature of the sample being sent through
            % (remember, i controls the sample being picked from the first
            % fold and j controls the jth feature of that sample) and the
            % kth sample and the jth feature the neuron in the pattern
            % layer. Remember k is picked first so k stays constant until
            % all the samples in the fold have gone through.
            temp_dist_controls = 0;

            % m iterates on the number of features of the input layer
            % because the Euclidean distance to a neuron in the pattern
            % layer involves summing the differences between each feature
            % of a neuron in the pattern layer and a feature of a sample in
            % the input layer.
            for m = 1:size(f1,2)-1
                temp_dist_controls = temp_dist_controls + (f1(l,m) -
controls(k,m)).^2./(2.*current_best_sigmas(1,m).^2);

            end
```

```
        % Store the Euclidean distance in the kth row (for the kth
        % neuron in the control class) and l column (this corresponds
        % to the sample in the input layer)
        % ---------l-----------  Along a column, these will be the
        % -                 -  Euclidean Distances from a sample
        % -                 -  in the input layer and kth neuron
        % k                 -  in the control class of the pattern
        % -                 -  layer and when this process is
        % ---------------------  repeated in the case class in the
        % columns will be compared because those are the distances
        % from the sample in the input layer to the neurons in the
        % pattern layer. This is why I store the output of
        % temp_dist_controls in an array called distCon in the kth row
        % and the l column.
        distCon(k,l) = temp_dist_controls;


    end
    % Move on to the next neuron in the control class of the pattern l
    % layer
    k = k + 1;


end


% k is reset to 1 and now iterates on the number of neurons on the
% case class of the pattern layer. The first sample is picked and the
% Euclidean Distances are calculated using that neuron
k = 1;

% a 'while' loop just picks the first sample to begin with
while k <= size(cases,1)

    % We now iterate on the number of samples in fold 1(the one that
    % was removed, starting from the first sample all the way to the
    % end. i is simply accessing the sample we are picking. If we are
    % considering a matrix, then you can picture having l lines of
    % samples within the fold
    for l = 1:size(f1,1)


        % We now initialize an accumulator variable to 0 because we
        % now have to calculate the Euclidean Distance and sum of the
        % differences between each feature of the neuron in the pattern
```

```
        % layer and the feature in the sample being sent through the
        % case class of the pattern layer. This value of the
        % Euclidean Distance is the difference between the ith sample
        % and the jth feature of the sample being sent through
        % (remember, i controls the sample being picked from the first
        % fold and j controls the jth feature of that sample) and the
        % kth sample and the jth feature the neuron in the pattern
        % layer. Remember k is picked first so k stays constant until
        % all the samples in the fold have gone through.
        temp_dist_cases = 0;

        % m iterates on the number of features in the sample that was
        % passed into the input layer. the accumuluator variable
        % temp_dist_cases adds up the differences between the feature
        % of the neuron in the case class of the pattern layer
        for m = 1:size(f1,2)-1
           temp_dist_cases = temp_dist_cases + (f1(l,m) -
cases(k,m)).^2./(2.*current_best_sigmas(1,m).^2);

        end

        % Store the Euclidean distance in the kth row (for the kth
        % neuron in the control class) and l column (this corresponds
        % to the sample in the input layer)
        % ---------l-----------  Along a column, these will be the
        % -                 -  Euclidean Distances from a sample
        % -                 -  in the input layer and the kth neuron
        % k                 -  in the case class of the pattern
        % -                 -  layer and when this process is
        % ---------------------  repeated in the control class the
        % columns will be compared because those are the distances
        % from the sample in the input layer to the neurons in the
        % pattern layer. This is why I store the output of
        % temp_dist_cases in an array called distCas in the kth row
        % and the l column.

        distCas(k,l) = temp_dist_cases;


    end
    % Move on to the next neuron in the case class of the pattern
    % layer
    k = k + 1;


end
```

```
% Apply the exponential function to each element in the arrays
% containing the distances from each sample in the input layer to all
% the neurons in the pattern layer
expDistCon =exp(-1.*distCon);
expDistCas = exp(-1.*distCas);

% p corresponds to the number of features
p = size(dat,2)-1;

% Summation layer calculating Fa and Fb where Fa is the sum of all the
% exponentiated distances from the controls class and Fb is the sum of
% all the exponentiated distances from the cases class.
Fa = (1./((2.*pi).^(p./2).*sigProd.*size(controls,1))).*sum(expDistCon);
Fb = (1./((2.*pi).^(p./2).*sigProd.*size(cases,1))).*sum(expDistCas);

% Defining an empty array will contain the classifications the PNN
% makes by comparing values from Fa and Fb at the same time. Fa and Fb
% will be 1 by 105 row vectors and we'll be comparing each
% value in the same columns of Fa and Fb. Assigning a 0 if Fa is bigger
% and assigning a 1 if Fb is greater.
assign = [];
for c = 1:size(Fa,2)

   if Fa(1,c) > Fb(1,c)
      assign = [assign;0];
   else
      assign = [assign;1];

   end
end



% Initialize an accumulator variable that will tally the number of
% correct classifications the PNN makes. The gold truths in the
% fold are compared to the classifications found in the 'assign' array
numCorr = 0;
for a = 1:size(assign,1)

   if assign(a,1)==f1(a,size(dat,2))
      numCorr = numCorr+1;
   end
end
```

```
   % Since we are finding percent accuracies for the given values of sigma
   % Each set of sigmas will yield a percent accuracy when fold 1 is
   % validated. So for each set of sigma I store a percent accuracy in a
   % column vector.
   percentAccuracy(i,1) = (numCorr./(size(f1,1))).*100;

end


end
```

pnnTrain

```
function[acc] = pnnTrain(n,sigmas,dataDim,controls,cases)

f_controls = [];
f_cases = [];
temp_dist_controls = []; %Null set
temp_dist_cases = [];
temp_acc = [];
acc = [];


for s = 1:size(sigmas,2)   %Outer for loop controls sigma we are on

   for j = 1:dataDim(1,2)-1   %3rd column is gold standard for data
                              %Columns 1 and 2 are the 2 features
      k = 1;

      while k <=size(controls,1)   %k iterates on the number of rows in controls
         %This while loop is essentially picking samples from controls
         %and being used in the calculation of the Euclidean Distances
         %to both the controls class itself and the cases class.
         %Remember it keeps incrementing by 1 to specify what row it is
         %in

%------------------------%------------------------%----------------------------------------
         for i = 1:size(controls,1)
            %Outer while looping through controls
            %Need inner for loop going through controls class again


            if k~=i %Check if the row while loop picked matches the row the for loop is in
               temp_dist_controls = [temp_dist_controls;...
                  exp(-1*((controls(k,j) - controls(i,j) ).^2/(2.*sigmas(1,s).^2)))];
```

```
            end
        end
 %-----------------%-----------------%-----------------------------------------------------

        for i =1:size(cases,1)  %Iterate through the cases and calculate distances
            %Negate the values and use exponential function on all
            %these values
            %Store in temporary array for control to case distances
            %only with the first column
            temp_dist_cases = [temp_dist_cases;...
                exp(-1*((controls(k,j) - cases(i,j) ).^2 ./ (2.*sigmas(1,s).^2)))];
        end
%-------------------------------------------------------------------------------------------

        f_controls = sum(temp_dist_controls)*...
            (1./ ((2.*pi).^(0.5).*(sigmas(1,s)).*size(temp_dist_controls,1)));
        %Control to control fa


        f_cases = sum(temp_dist_cases)*...
            (1./(((2.*pi).^(0.5).*sigmas(1,s).*size(temp_dist_cases,1))));
        %control to case fb


        temp_dist_controls = [];
        temp_dist_cases = [];

        if f_controls>f_cases
            temp_acc = [temp_acc;1];  %Assign a 1 to the accuracy list if controls was
classified correctly


        elseif f_controls<f_cases
            temp_acc = [temp_acc;0];
            %Assign a 0 if sample from controls was classified
            %incorrectly
        end

        temp_dist_controls = [];
        temp_dist_cases = [];
        f_controls = [];
        f_cases = [];
```

```
        k = k + 1; %Move on to the next row by incrementing k by 1


    end



    %-------Now the program will pick samples from cases------%

    k = 1;%Counter will keep track of rows in controls since we use a while loop

    while k <= size(cases,1)


%-------------------------%-------------------------%----------------------------------------
        for i = 1:size(cases,1)
          %Outer while looping through controls
          %Need inner for loop going through controls class again


          if k~=i %Check if the row while loop picked matches the row the for loop is in
             temp_dist_cases = [temp_dist_cases;...
                exp(-1*((cases(k,j) - cases(i,j) ).^2 ./ (2.*sigmas(1,s).^2)))];

          end

        end
 %------------------%------------------%-----------------------------------------------------

        for i =1:size(controls,1)  %Iterate through the cases and calculate distances
          %Negate the values and use exponential function on all
          %these values
          %Store in temporary array for control to case distances
          %only with the first column
          temp_dist_controls = [temp_dist_controls;...
             exp(-1*((cases(k,j) - controls(i,j) ).^2 ./ (2.*sigmas(1,s).^2)))];
        end
%-------------------Construct the probability density functions knowing ha and la are 1-----
----------

        f_controls = sum(temp_dist_controls).*...
          (1/((2*pi).^(0.5).*(sigmas(1,s)).*size(temp_dist_controls,1)));
        %Density function for case to control distances


        f_cases = sum(temp_dist_cases).*...
```

```
            (1./((2.*pi).^(0.5).*(sigmas(1,s)).*size(temp_dist_cases,1)));
          %Density function for case to case distances

%-------------------------------------------------------------------------------------------------

          if f_controls>f_cases
             temp_acc = [temp_acc;0];  %Assign a 0 if the network shows that the sample
belongs to the controls class


          elseif f_controls<f_cases
             temp_acc = [temp_acc;1]; %Assign a 1 if the network shows that the sample
belongs to the case class
             %Assign a 0 if sample from controls was classified
             %incorrectly
          end

          temp_dist_controls = [];
          temp_dist_cases = [];
          f_controls = [];
          f_cases = [];



       k = k +1;

       end

       for i = 1:size(temp_acc,1)  %Temp accuracy is a column vector with rows
          x = sum(temp_acc == 1);
       end

       acc(s,j) = (x / ( size(temp_acc,1))).*100; %Calculate percent accuracy
                                          %and store in final
                                          %accuracy matrix.

       temp_acc = []; %Make the temp acc array empty ...
                ...again because we will move on to the next feature


    end


end

acc;
```

```
end
```

EP/ES SCRIPTS AND FUNCTIONS

Create_intitial_population

```
function [gen_0_sigmas] = create_initial_population(nPopulation,nFeatures)
%This function uses an Evolutionary Programming Technique to generate
%sigma values for each feature and mutates them in a pseudo-random manner
%to create a set of mutated sigmas. The set of 0 generation sigmas and 1st
%generation sigmas allows for increased search space to find the best
%values of sigmas that return the highest acccuracy and the highest Az
%value when doing an ROC analysis. The output of the function contains the
%0 generation sigmas and the mutated (1st generation) sigmas.


%Creating initial population


% From senior design report...
min_init_sigma = 0.005;
max_init_sigma = 0.0055;

sigmas = zeros(25,12);
for i = 1:nPopulation %For each population


   for j = 1:nFeatures %For each feature...
      sigmas(i,j) = random('Uniform',min_init_sigma,max_init_sigma);
   end

end

gen_0_sigmas = sigmas;


end
```

Mutate_sigmas

```
function [gen_next_sigmas] = mutate_sigmas(nPopulation,nFeatures,gen_0_sigmas)

mutated_sigmas = zeros(25,12); % Pre-allocate 25x12 matrix of zeroes for
```

```
                    % mutated-sigma values

for i = 1:nPopulation %For each population in the complete data set

   xi_list = zeros(1,12);
   N = random('Normal',0,1); % Store the value of a random variable taken from a
Normal Distribution for the population

   V = zeros(1,12);
   %Create a random cauchy variable...
   cauch = random('Uniform',-4,4); %  Random value between -4 and 4 to create a cauchy
variable
   C = 1./(pi.*(1+cauch.^2)); %Initialize random normal variable for population

   for j = 1:nFeatures %For each feature in the data set
      V_i = random('Uniform',0.009,0.02); %Generate mutation element
      V(1,j) = V_i; %Add mutation element to mutation vector
      N_prime = random('Normal',0,1); %Initialize random normal variable for each
feature
      xi = exp( (1./(((2.*nFeatures).^.5)).*N) + ( (1./((2 .*(nFeatures.^.5)).^.5)
.*N_prime)) ); %Math of mutation
      xi_list(1,j) = xi; %Add xi to xi list
   end

   xi = xi_list;
   V_prime = V.*xi; %Multiple xi and mutations, Element wise multiplication of 2 arrays
   mutated_sigmas(i,:) = gen_0_sigmas(i,1:12)+(C.*V_prime);


end



gen_next_sigmas = [gen_0_sigmas;mutated_sigmas];


end
```

PNN_Training

```
function [sigmas_and_accs] = PNN_Training(sigmas,controls,cases)


% Let controls be represented as a
% Let cases be represented as b
```

```
for i = 1:size(sigmas,1)

   current_sigmas = sigmas(i,:); %Pick the set of sigmas on the row we are on

   %Set an initial variable to 1 so the sigma product can be calculated
   %Sort of works like an accumulator variable
   sigProd = 1;

   % For the current set of sigmas that we are on keep multiplying the
   % value in sigProd by the sigma value in the sigma set that was picked
   % out by the iterator i. j will iterate on all the values of sigma and
   % calculate a total product by mutiplying sigprod by the next value of
   % sigma in the set picked out.
   for j = 1:size(current_sigmas,2)
      sigProd = sigProd.*current_sigmas(1,j);
   end

   k = 1;
   while k<=size(controls,1) % Start with a neuron in the pattern layer

      for l = 1:size(controls,1) % Pick sample

         temp_dist_controls = 0;

         for m = 1:size(controls,2)-1 %Every feature of sample
            if k~=l
               temp_dist_controls = temp_dist_controls+...
                  (controls(l,m) - controls(k,m)).^2./(2.*current_sigmas(1,m).^2);
            end
         end

         a_to_a(k,l) = temp_dist_controls; % Control-Control Distances

      end


      for l = 1:size(cases,1) % For each neuron in the case class of pattern layer


         temp_dist_cases = 0;

         for m = 1:size(cases,2)-1 % For each feature of case neuron in pattern layer

            temp_dist_cases = temp_dist_cases+...
               (cases(l,m)-controls(k,m) ).^2 ./ (2.*current_sigmas(1,m).^2);
```

```
        end

        a_to_b(k,l) = temp_dist_cases; %Control to Case Distances

    end

    k = k+1;
end



k = 1;

while k <= size(cases,1)

    for l = 1:size(controls,1)

        temp_dist_controls = 0;

        for m = 1:size(controls,2)-1

            temp_dist_controls = temp_dist_controls+...
                (controls(l,m)-cases(k,m) ).^2 ./ (2.*current_sigmas(1,m).^2);
        end

        b_to_a(k,l) = temp_dist_controls;
    end

    for l = 1:size(cases,1)
        temp_dist_cases = 0;

        for m = 1:size(cases,2)-1
            if k~=l
                temp_dist_cases = temp_dist_cases+...
                    (cases(l,m)-cases(k,m) ).^2 ./ (2.*current_sigmas(1,m).^2);
            end
        end
        b_to_b(k,l) = temp_dist_cases;

    end


    k = k+1;
end

a_to_a = exp(-1.*a_to_a);
```

```
a_to_b = exp(-1.*a_to_b);
b_to_a = exp(-1.*b_to_a);
b_to_b = exp(-1.*b_to_b);

p = size(controls,2)-1;

f_a_to_a = sum(a_to_a,1)*...
   (1./ ((2.*pi).^(p/2).*(sigProd).*size(controls,1)));
%Control to control f


f_a_to_b = sum(a_to_b,1)*...
   (1./(((2.*pi).^(p/2).*sigProd.*size(cases,1))));
%control to case f

f_b_to_a = sum(b_to_a,1)*...
   (1./ ((2.*pi).^(p/2).*(sigProd).*size(controls,1)));
%Control to case f

f_b_to_b = sum(b_to_b,1)*...
   (1./ ((2.*pi).^(p/2).*(sigProd).*size(cases,1)));
%Case to control f

% Make Classifications
assign = [];
for c = 1:size(f_a_to_a,1)
   if f_a_to_a(c,1)>f_a_to_b(c,1)
      assign = [assign;0]; % Assign a 0 for benign cases
   elseif f_a_to_a(c,1) < f_a_to_b(c,1)
      assign = [assign;1];
   end
end

for d = 1:size(f_b_to_a,1)
   if f_b_to_b(d,1) > f_b_to_a(d,1)
      assign = [assign;1]; % Assign a 1 for malignant cases
   elseif f_b_to_b(d,1) < f_b_to_a(d,1)
      assign = [assign;0];
   end
end

%    numCorr = 0;
%    for e = 1:size(assign,1)
%       if assign(e,1)==1
%          numCorr = numCorr+1;
%       end
```

```
%     end

    complete = [controls;cases];

    numCorr = 0;
    for e = 1:size(assign,1)
        if assign(e,1)==complete(e,13)
            numCorr = numCorr+1;
        end
    end

    one_acc = (numCorr./(size(complete,1)-1)).*100;

    train_acc(i,1) = one_acc;

end

sigmas_and_accs = [sigmas,train_acc];




end
```