EEMM055 - AI and AI Programming Assignment
MATLAB simulation using the Neural Network Toolbox

Jamie Dance - URN: 6661320

NN MATLAB simulation for ‘Understanding how to solve pattern recognition problems using backpropagation’

UNIVERSITY OF SURREY

Contents

Executive Summary

This report includes the analysis and findings of a pattern recognition neural network problem using backpropagation. The aim is to build a model that can predict whether a cancer patient's condition is 'benign' or 'malignant'. There are 699 instances each with a value of 1-10 over a series of 9 variables and a value of 1 or 0 for an output which determines whether a cancer patients' condition is or isn't malignant respectively. This report explores the benefit of finding the optimal value of epochs and nodes, the benefit or detriment of using ensembled methods for classifiers and the comparison of 3 different training optimisers.

Throughout this report we will be using Error Rate as a measure of our classification performance. Error Rate = number of misclassifications divided by total number of patterns (fraction or percentage). This is because Error Rate is the best universal way to evaluate the performance of multiple different models at the same time.

In this report it was found that for the breast cancer dataset the best combination of epoch and nodes was 8 and 8. This provided us with the best test Error Rate accuracy at 0.0178 (1.78%) for the patternnet neural network using the 'trainscg' optimiser. Next using this optimal value of nodes, we explored the use of ensembled classifiers over 4, 8 and 16 epochs. Our results found that the ensembled classifiers performed significantly better at lower values of epochs but performed significantly worse at higher values of epochs. In addition, when using 4 and 8 number of epochs there was a clear descent in the gradient of Error Rate. This suggests that increasing the number of classifiers in an ensemble benefits the prediction at these levels of epochs. On the other hand, there

was near to no descent or change with the addition of classifiers at 16 epochs. This likely suggests that there is little to no room to further improve the accuracy of the model at 16 epochs and there is no need to increase the number of classifiers in the ensemble. Lastly, it was found that 'trainscg' and 'trainrp' performed the best of the 3 training optimisers, with trainscg taking the slight edge over trainrp as the number of classifiers increased. Although, it is important to mention that in our analysis we were using optimal number of nodes and epochs for trainscg found in the previous exercises. If I had more computational power, I would have explored the effect of increasing and decreasing the number of epochs for each training optimiser. In conclusion using the base classifier, 8 epochs and 8 nodes we have developed a pattern recognition model with an error rate of 1.78%. This would suggest that our model is very accurate and is a viable method of predicting whether a patient's condition is 'benign' or 'malignant'.

Introduction

In this report we will be developing a neural network MATLAB simulation with the aim to understand how to solve a pattern recognition problem using backpropagation. Backpropagation, short for "backward propagation of errors," is an algorithm for supervised learning of artificial neural networks using gradient descent. Given an artificial neural network and an error function, the method calculates the gradient of the error function with respect to the neural network's weights. 'It is a generalization of the delta rule for perceptrons to multilayer feedforward neural networks.' (Backpropogation, n.d.).

In this example we will be using the breast cancer dataset obtained from the University of Wisconsin Hospitals. There is a total of 699 number of instances and 10 attributes (9 inputs and 1 output/class). The inputs range are as follows: Clump Thickness, Uniformity of Cells Size, Uniformity of Cell Shape, Marginal Adhesion, Single Epithelial Cell Size, Bare Nuclei, Bland Chromatin, Normal Nucleoli and Mitoses. Each of these inputs range from a value of 1 to 10. Using these inputs and pattern recognition software such as the MATLABS' "Neural Network Pattern Recognition" app we will aim to develop a program that can determine whether a future patient's condition is 'benign' or 'malignant'.

Over the duration of this report we will test different methods of obtaining better accuracy for our model. The methods we will be looking at is parameter optimisation, namely epochs and hidden layers (nodes), ensemble classifier methods and different training algorithms.

For Exp 1) we will be running our pattern recognition neural network with a variety of different epoch and node (hidden layer) values. For each node value, we will plot a line graph of Error Rate vs Epochs. Using the graph and the statistics found in our simulation we will determine the optimal value for the test error rate and the associated epoch/node values.

For Exp 2) we will explore the use of an ensemble of individual classifiers to try improving test accuracy. I will be using the optimal value of nodes found in Exp 1) and testing it over a range of 3 epochs. Each ensemble will be plotted with test Error Rate vs Number of Classifiers, this will let us understand if using larger ensembles of classifiers is beneficial to improve accuracy.

For Exp 3) we will be comparing the performance of 3 different training optimisers: 'traingscg' 'trainlm' and 'trainrp'. Each optimiser uses a variant of backpropogation and I will explore which provides the best accuracy for our dataset.

Exercises

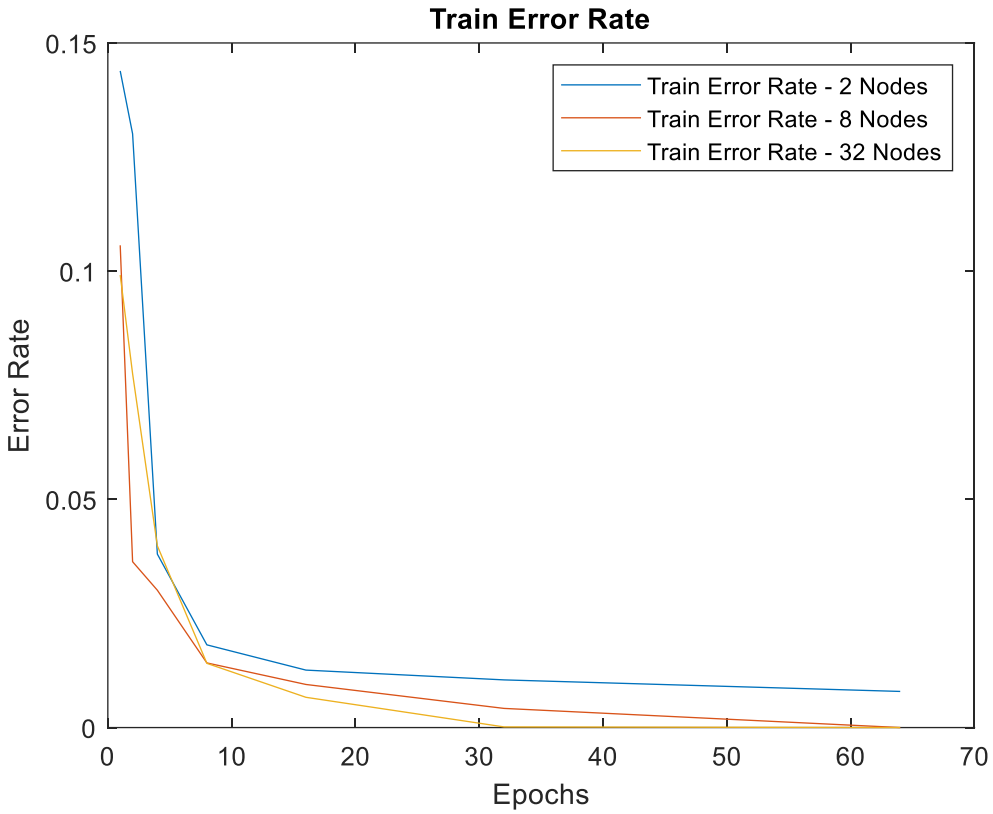
Exp 1)

For this report we will be using the matlab neural network function “patternnet” which generates a pattern recognition neural network. Pattern recognition is the process of training a neural network to assign the correct target classes to a set of input patterns. Once trained the network can be used to classify patterns it has not seen before. Pattern recognition networks are feedforward networks that can be trained to classify inputs according to target classes.

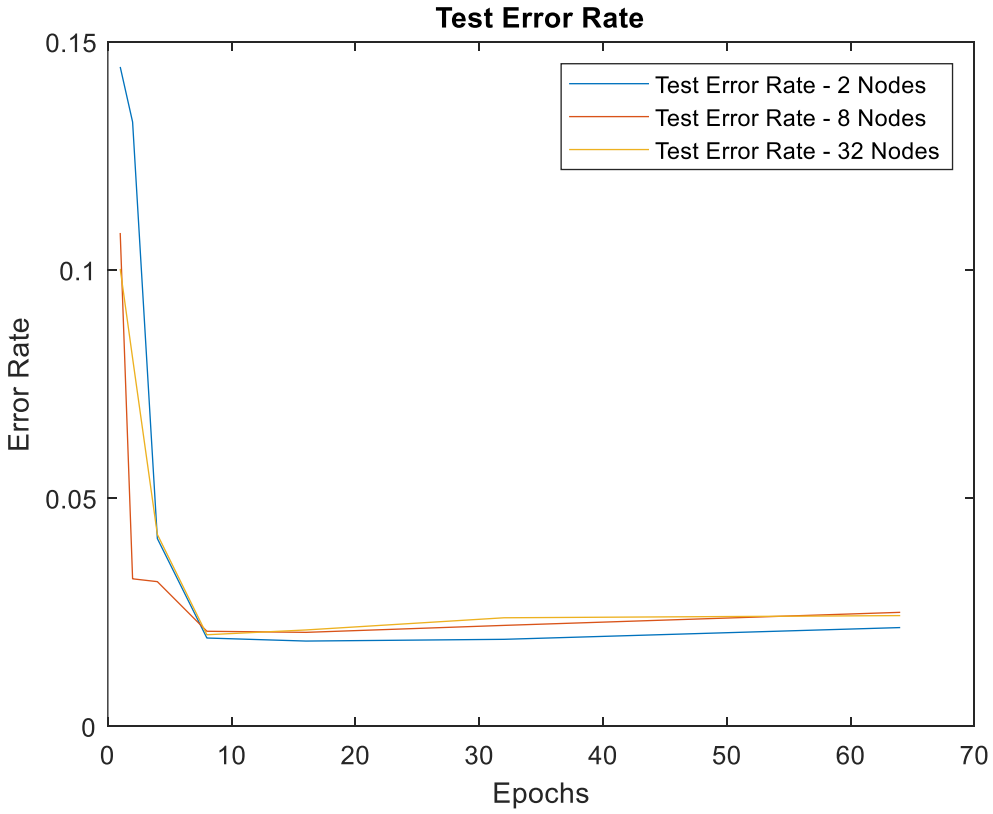
The epochs parameter determines the maximum number of training iterations, that refers to one cycle through the full training dataset. In other words, if we feed a neural network the training data for more than one epoch in different patterns, we hope for a better generalisation when given the test data. Hidden layers (nodes) determines the number of neurons in the hidden layer. In general, increasing the number of neurons in the hidden layer increases the power of the network, but requires more computation and is more likely to produce overfitting.

We will be using a 50/50% split for train and test data. This means we will not be implementing a validation set which could cause us to have some overfitting issues when we test the model. As a result, it is important that we make close comparisons between the performance of the train dataset and the test dataset. If there is a clear difference between the two at a certain node/epoch combination it could suggest overfitting.

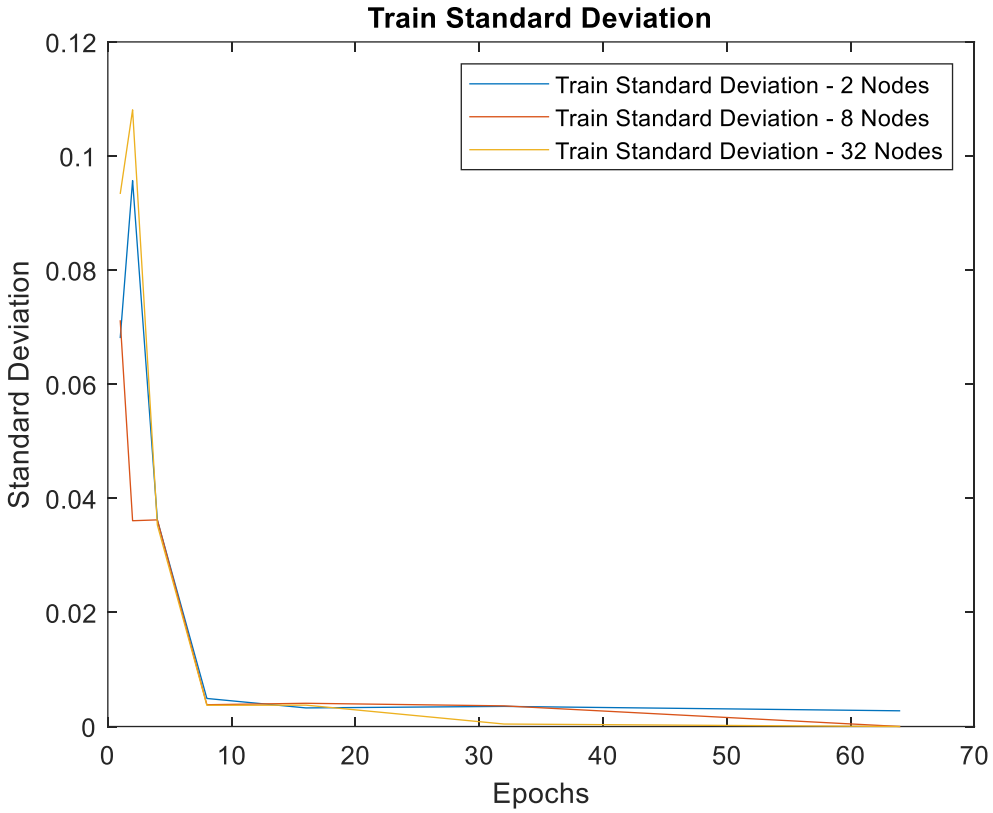
We will be running our pattern recognition simulation 30 times for each combination of epoch and node then calculating the average. This is to receive a better result as it is likely that each model may make slightly different predictions, and when evaluated using Error Rate, may have a slightly different performance.



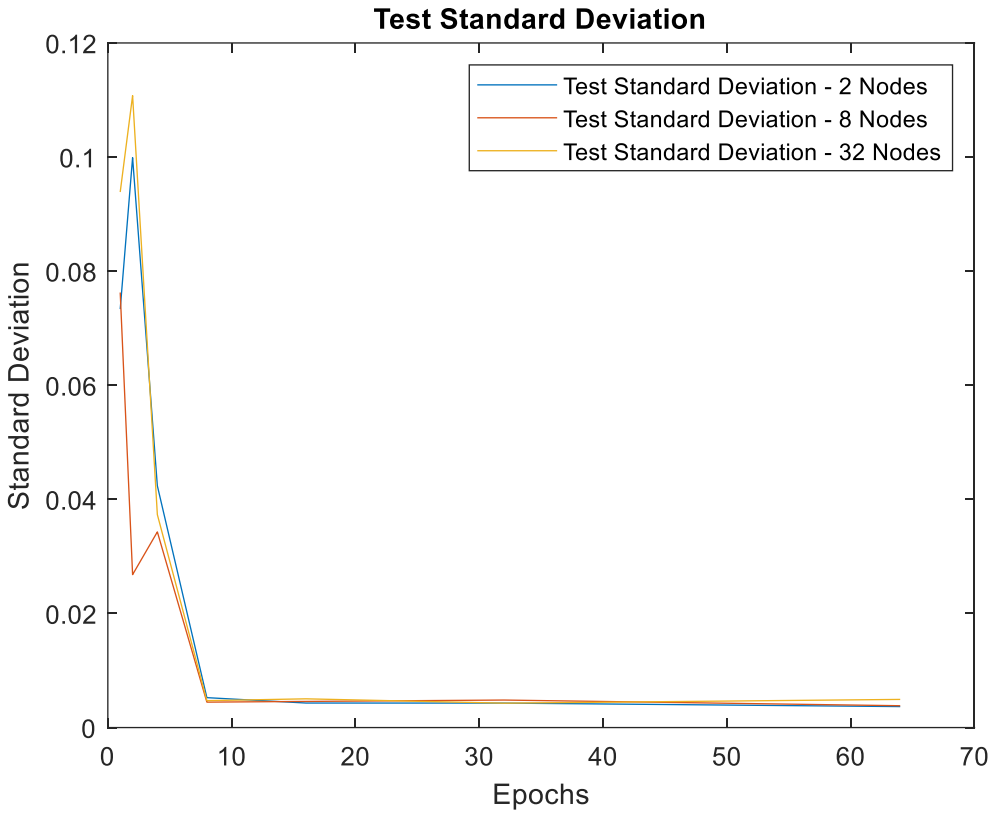
We can see that train Error Rate for each node value drastically drops till 8 epochs, after that it only slowly descends, and it appears that each line respective to their nodes converges to 0. The 32-node line appears to descend quicker than the other 2 and hits 0 much sooner at roughly 32 epochs.



We can see that test Error Rate for each node value drastically drops till 8 epochs where the gradient smooths out and is consistent at roughly 0.025. In fact, it actually appears that the Error Rate slightly increases after 8 epochs but not by a significant amount. The value for our test Error Rate is similar to our Train Error rate which suggests that there is no overfitting in our model.



We can see that train standard deviation for each node value drastically drops till 8 epochs, after that it only slowly descends, and it appears that each line respective to their nodes converges to 0. The 32-node line appears to descend quicker than the other 2 and hits 0 much sooner at roughly 32 epochs.



We can see that test standard deviation for each node value drastically drops till 8 epochs where the gradient smooths out and is consistent at roughly 0.005.

```
Optimal =

  2×6 string array

    "Nodes"    "Epoch"    "AvgErrorTrain"    "AvgErrorTest"    "StdErrorTrain"    "StdErrorTest"
    "8"        "8"        "0.015546"         "0.017883"        "0.0060432"        "0.0039179"
```

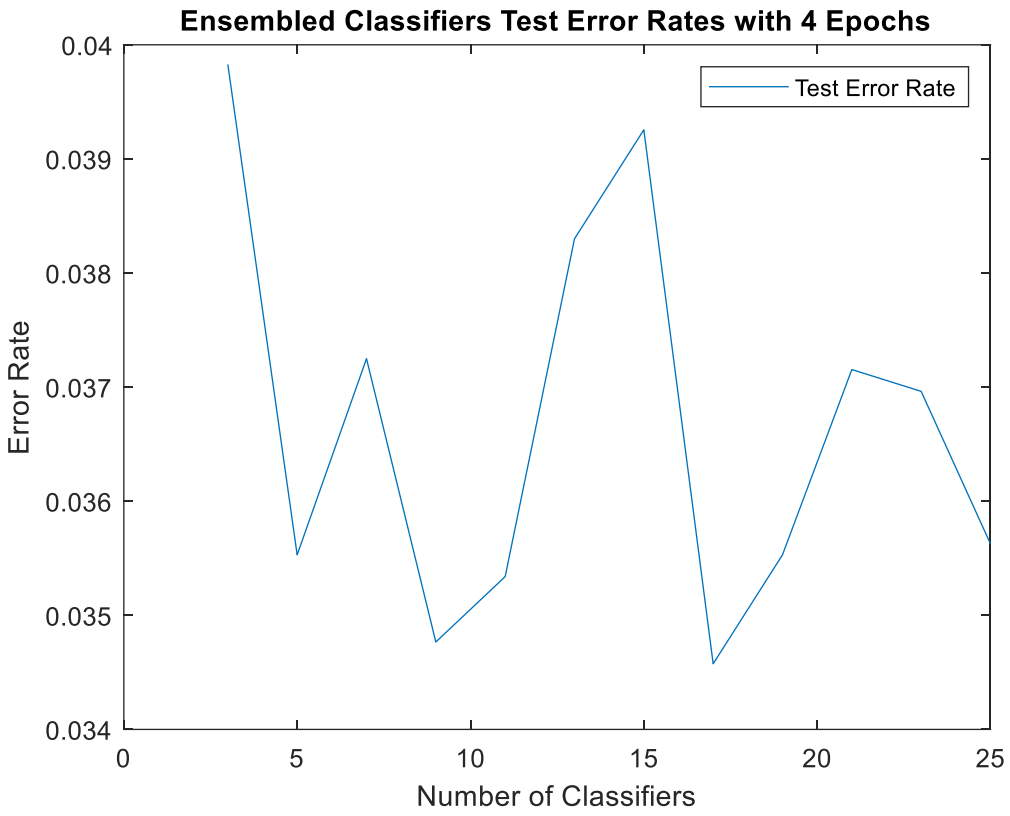
As suggested by every graph and the table above the optimal value of nodes is 8 and the optimal value of epochs is 8. This means that the optimal number of neurons within the hidden layer are 8 and the optimal amount of iterations to run our simulation is also 8.

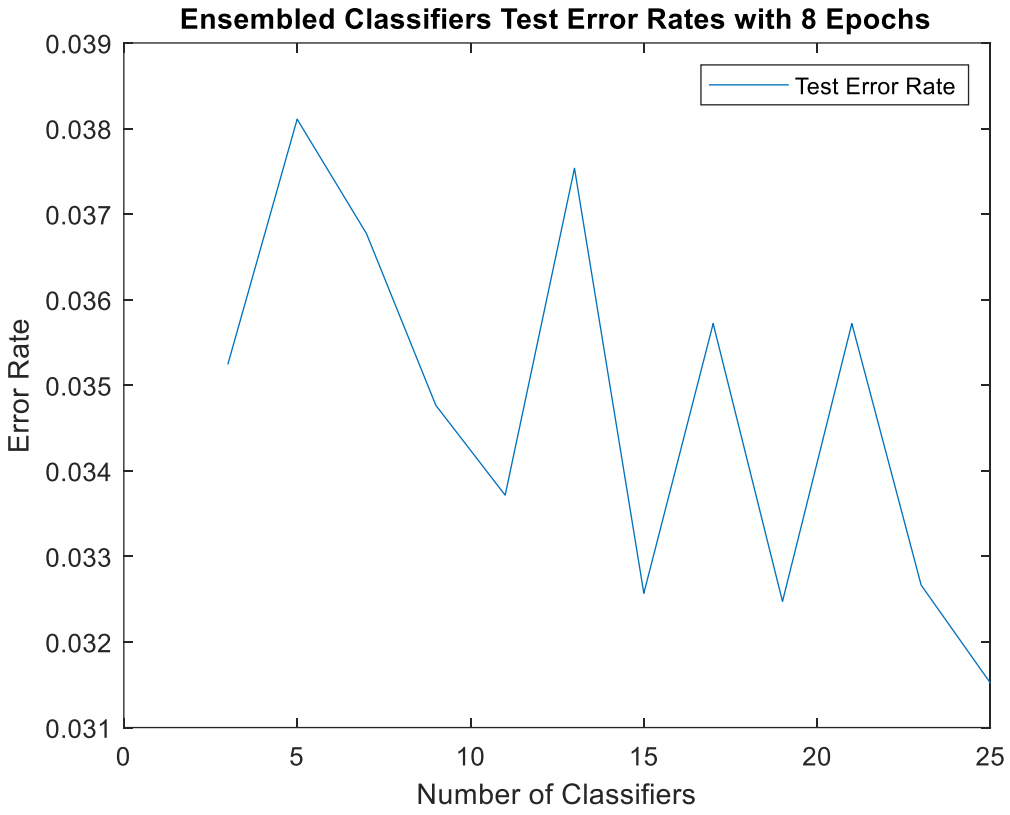
Exp 2)

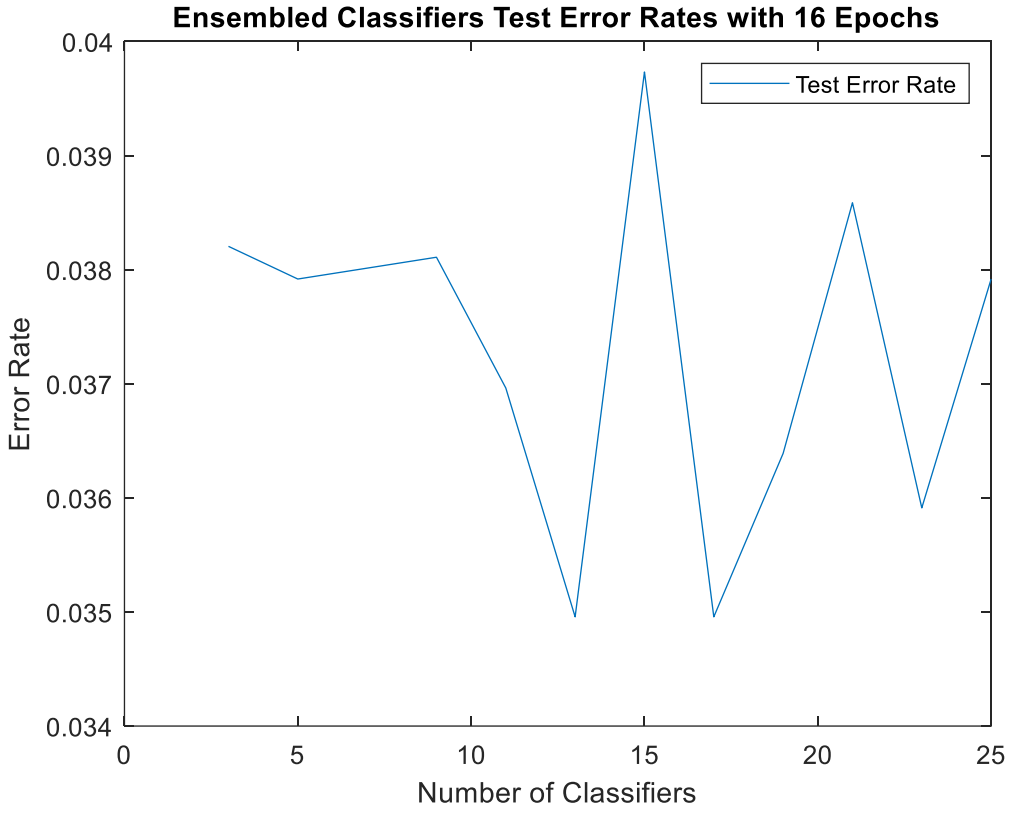
Deep learning neural networks are nonlinear methods. They offer increased flexibility and can scale in proportion to the amount of training data available. A downside of this flexibility is that they learn via a stochastic training algorithm which means that they are sensitive to the specifics of the training data and may find a different set of weights each time they are trained, which in turn produce different predictions. Generally, this is referred to as neural networks having a high variance and it can be frustrating when trying to develop a final model to use for making predictions. A successful approach to reducing the variance of neural network models is to train multiple models instead of a single model and to combine the predictions from these models. This is called ensemble learning and not only reduces the variance of predictions but also can result in predictions that are better than any single model. Ensemble learning is a machine learning technique that combines several base models in order to produce one optimal predictive model. An ensemble of classifiers is a set of classifiers whose individual decisions are combines in some way (typically by weighted or unweighted voting) to classifier new examples.

In hard voting (also known as majority voting), every individual classifier votes for a class, and the majority wins. In statistical terms, the predicted target label of the ensemble is the mode of the distribution of individually predicted labels. Every model makes a prediction (votes) for each test instance and the final output prediction is the one that receives more than half of the votes. If none of the predictions get more than half of the votes, we may say that the ensemble method could not make a stable prediction for this instance. Although this is a widely used technique, you may try the most voted prediction (even if that is less than half of the votes) as the final prediction. In this exercise we will be using an odd number of classifiers (25) so that there can be no tie for the majority vote.

In general, from the graphs produced of the ensembled classifiers there is not a clear improvement when compared with the base classifier. At lower values of epochs the ensembled classifiers appears to perform slightly better and shows a general descent as more classifiers are added. This could suggest that once our model hits a certain amount of iterations (8) there is not much room for further improvement. This is also suggested from the previous exercise as the average test error rate and average standard deviation appears to not change past 8 epochs. This further backs up our assumption that 8 epochs are the optimal amount. Although at higher values of epochs (16) it appears that the ensembled classifier performed worse than the base classifier.




Ensembled Classifiers Test Error Rates with 8 Epochs
Test Error Rate
Error Rate
Number of Classifiers
0.039
0.038
0.037
0.036
0.035
0.034
0.033
0.032
0.031
0
5
10
15
20
25


Ensembled Classifiers Test Error Rates with 16 Epochs
Test Error Rate
Error Rate
Number of Classifiers
0.04
0.039
0.038
0.037
0.036
0.035
0.034
0
5
10
15
20
25

Exp 3)

'trainscg' - Scaled conjugate gradient backpropogation:

trainscg is a network training function that updates weight and bias values according to the scaled conjugate gradient method. It can be used to train any network as as its weight, net input, and transer functions have derivative functions. Backpropogation is used to calculate derivatives of performance 'perf' with respect to the weight and bias variables x. trainscg can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X.

The scaled conjugate gradient algorithm is based on conjugate directions, as in 'trainchp', 'traincgf' and 'traincgb', but this algorithm does not perform a line search at each iteration.

'trainlm' - Levenberg-Marquardt backpropagation:

'trainlm is a network training function that updates weight and bias values according to Levenberg-Marquardt optimisation. It is often the fastest backpropogation algorithm and is highly recommended as a first-choice supervised algorithm, although it does require more memory that other algorithms. trainlm can train any network as long as its weight, net input, and transfer functions have the same derivative functions.

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares, the the Hessian matix can be approximated as:

$$H = J^T J$$

and the gradient can be computed as:

$$g = J^T e$$

where J is the Jacobian matrix that contains the first derivatives of the network errors with respect to the weights and biases, and e is a vector of the network errors.

'trainrp' - Resilient backpropagation:

trainrp is a network training function that updates weight and bias values according to the resilient backpropogation algorithm (Rprop). trainrp can train any network as long as its weight, net input, and transfer functions have derivative functions. rprop is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. You do need to store the update values for each weight and bias, which is equivalent to storage of the gradient.

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called "squashing" functions, because they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slopes must approach zero as the input gets large. This causes a problem when you use steepest descent to train a multilayer network with sigmoid functions, because the gradient can have a very small magnitude and, therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

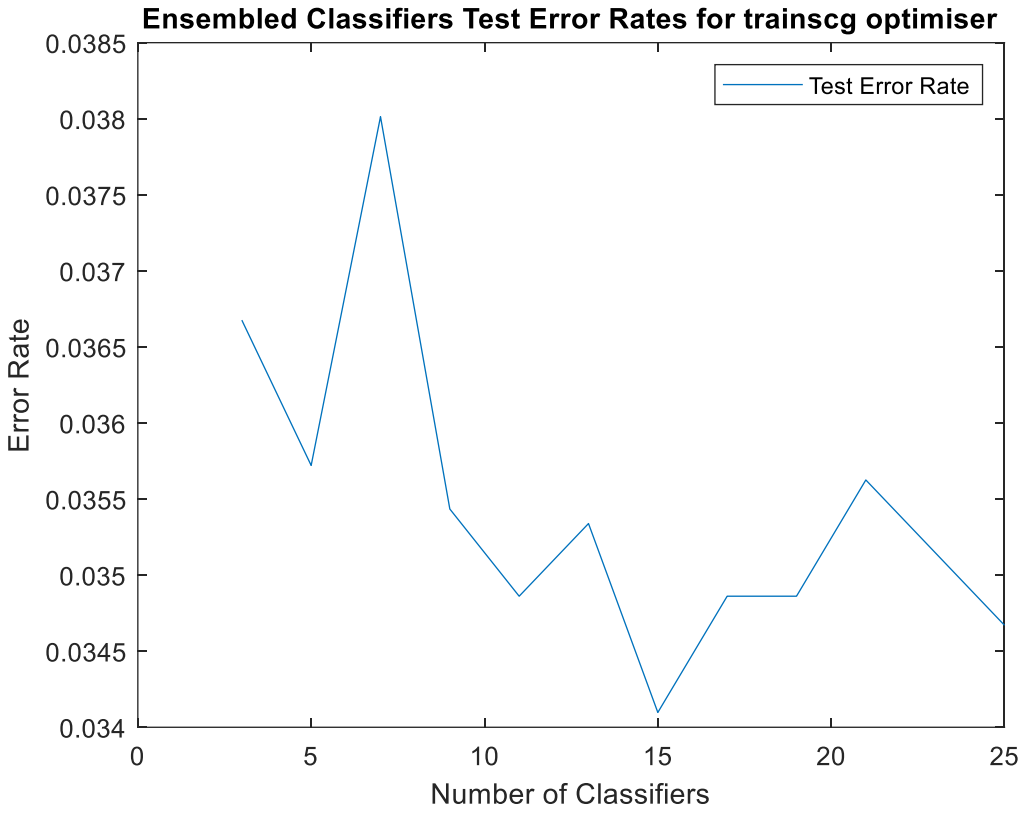
The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative can

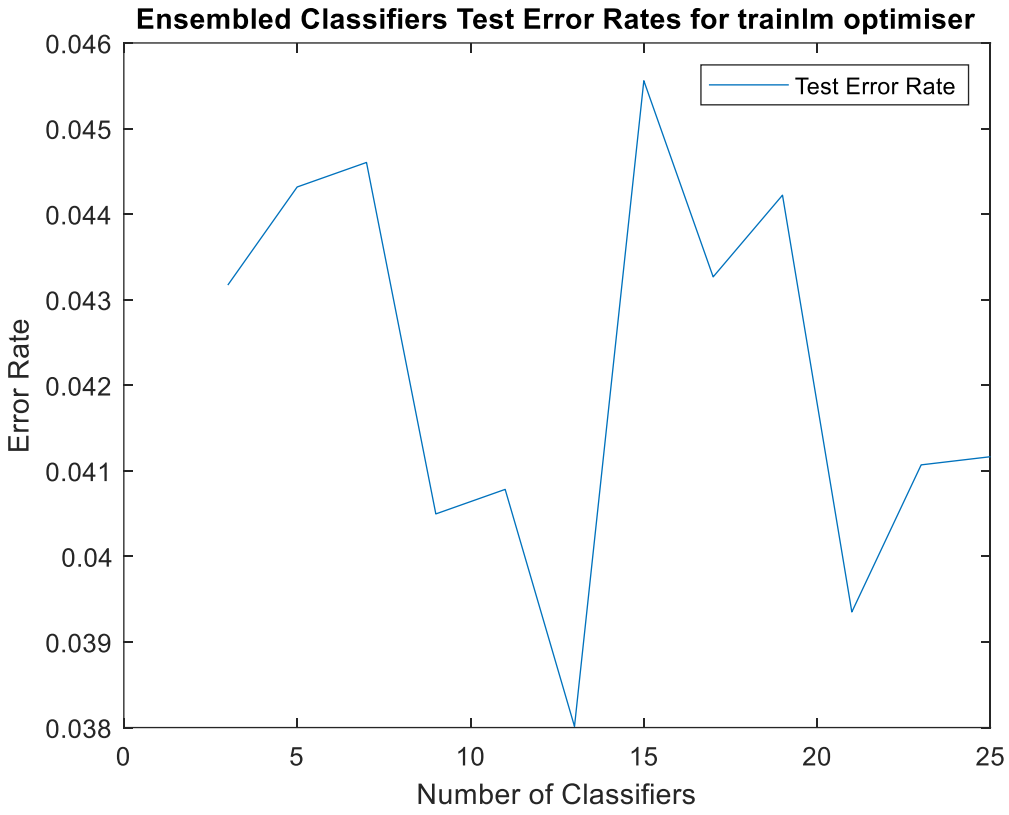
determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor delt_inc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor delt_dec whenever the derivative with respect to that weight changes sign from the previous iteration. If the derivative is zero, the update value remains the same. Whenever the weights are oscillating, the weight change is reduced. If the weight continues to change in the same direction for several iterations, the magnitude of the weight change increases.

Results:

As seen in the graphs for each training optimiser we can see that there is no particularly noticeable pattern of descent as classifiers increase for trainlm and trainrp. However, we can see that trainscg appears to have a consistent gradient descent. This suggests that the more classifiers we add to our ensemble we can expect the Error Rate to lower when using the trainscg optimiser.

Out of the 3 optimisers it appears trainlm performed the worst with a maximum value of Error Rate of around 0.046 (at 15 classifiers) and a minimum value of around 0.038 (at 13 classifiers). The minimum value of trainlm is equal to the maximum value of both trainscg and trainrp.




Ensembled Classifiers Test Error Rates for trainlm optimiser
Test Error Rate
Error Rate
Number of Classifiers
0.046
0.045
0.044
0.043
0.042
0.041
0.04
0.039
0.038
0
5
10
15
20
25


Ensembled Classifiers Test Error Rates for trainrp optimiser
Test Error Rate
Error Rate
Number of Classifiers
0.039
0.038
0.037
0.036
0.035
0.034
0.033
0
5
10
15
20
25

Conclusions

In summary when training a backpropagation pattern recognition neural network simulation to determine whether a patient's condition is 'benign' or 'malignant' the optimal number of epochs are 8 and the optimal number of nodes within the hidden layer is 8. Unless the user is required to use less than 8 epochs in their analysis it is not suggested to implement and ensemble of classifiers. Our findings suggested that the accuracy of our ensembled classifiers at 16 epochs was significantly worse than the base classifier. Alternatively, if the user implements less than 8 epochs our findings suggest that using an ensemble of classifiers is beneficial. Furthermore, adding more classifiers to the ensemble improves accuracy further. Finally, our findings suggested that the 'trainscg' or 'trainrp' are the best training optimisers to use for this problem. I would give trainscg the slight edge over trainrp as there is a clearer gradient decline as the number of classifiers are added. 'trainlm' should be avoided for this problem as it reported significantly worse results than the other 2 optimisers.

If I had more time and more computational power, I would explore every epoch value from 6 – 20 as this seems to be the best range for error rate accuracy. Subsequently I would explore the performance of the ensembles as they performed better at them lower values of epochs. Also, for Exp 3) I would plot at least 3 different values of epochs on the graphs for each training optimiser. This would allow to us to see if each training optimiser performed better at a different value of epochs rather than optimal amount of 8 found for just trainscg.

The dataset only consisted of 699 inputs which some people would consider a low amount. In my opinion for better accuracy more data should be collected to increase the validity of our outputs.

Lastly, I was unable to complete Exp 4) and if I had another chance at the project, I would seek more help in completing it.

Appendix

Exp1)

```
clear;

% Load Breast Cancer Dataset
load cancer_dataset;
% Define Inputs and Outputs
x = cancerInputs;
t = cancerTargets;

% Define the Training Functin
trainFcn = 'trainscg'; % using "trainscg" Train Function
% Define nodes(hidden layers) and epochs to use in analysis
nodes = [2,8,32];
trainEpochs = [1,2,4,8,16,32,64];
% Create titles for output table
Statistics =
["Nodes","Epoch","AvgErrorTrain","AvgErrorTest","StdErrorTrain","StdErrorTe
st"];

for n = nodes
    fprintf('\n%d Hidden Layers:\n',n)
    for e = trainEpochs
        fprintf('\n%d Epochs:\n',e)
        % Define the Neural Network, Epochs and Hidden layers
        hiddenLayerSize = n;
```

```
        net = patternnet(hiddenLayerSize, trainFcn);
        net.trainParam.epochs = e;

        % Choose Input and Output Pre/Post-Processing Functions
        net.input.processFcns = {'removeconstantrows','mapminmax'};

        %Setup Division of Data for Training, Testing
        net.divideFcn = 'dividerand';  % Divide data randomly
        net.divideMode = 'sample';  % Divide up every sample
        net.divideParam.trainRatio = 50/100;
        net.divideParam.valRatio = 0/100;
        net.divideParam.testRatio = 50/100;

        % Choose Confusion Plot Function
        net.plotFcns = {'plotconfusion'};

        % Train 30 neural networks
        numNN = 30;
        nets = cell(1, numNN);
        trs = cell(1, numNN);
        for i = 1:numNN
            % Train the nets
            fprintf('Training %d/%d\n', i, numNN)
            [nets{i},trs{i}] = train(net, x, t);

            % Calculate the predicted classification
            y = nets{i}(x);
            % Calculate Train and Test Targets
            trainTargets = t .* trs{i}.trainMask{1};
            testTargets = t .* trs{i}.testMask{1};
            trainTargetsy = y .* trs{i}.trainMask{1};
            testTargetsy = y .* trs{i}.testMask{1};
            % Change the vectors to index
            trainind = vec2ind(trainTargets);
            testind = vec2ind(testTargets);
            trainindy = vec2ind(trainTargetsy);
            testindy = vec2ind(testTargetsy);
            % Calculate Error Rate percentage for Train and Test
            percentErrorsTrain{i} = sum(trainind ~=
trainindy)/numel(trainind);
            percentErrorsTest{i} = sum(testind ~= testindy)/numel(testind);
        end
        % Change format of Error Rate Cell to double
        percentErrorsTrain2 = cell2mat(percentErrorsTrain);
        percentErrorsTest2 = cell2mat(percentErrorsTest);
        % Calculate the Average Error Rate
        percentErrorsAverageTrain = mean(percentErrorsTrain2);
        percentErrorsAverageTest = mean(percentErrorsTest2);
        fprintf('\nAverage Error Rate for Train: %4.5f\n',
percentErrorsAverageTrain)
        fprintf('Average Error Rate for Test: %4.5f\n',
percentErrorsAverageTest)
        % Calculate the Standard Deviation of Error Rate
        percentErrorsStdTrain = std(percentErrorsTrain2);
        percentErrorsStdTest = std(percentErrorsTest2);
        fprintf('\nStandard Deviation of Error rate for Train: %4.5f\n',
percentErrorsStdTrain)
        fprintf('Standard Deviation of Error rate for Test: %4.5f\n',
percentErrorsStdTest)
        % Add Average and Standard Deviation of Error rate to Statistics
```

```
        % table
        statadd =
[n,e,percentErrorsAverageTrain,percentErrorsAverageTest,percentErrorsStdTra
in,percentErrorsStdTest];
        Statistics(end+1,:) = statadd;
    end
end

% Convert to double format
StatisticsNum = str2double(Statistics([2:end],:))

% Optimal Value for Test Error Rate and Associated node/epoch values
[row, column] = min(StatisticsNum([1:end],4));
Optimal = Statistics([1,column+1],:)

%Plots
%Confusion Matrix
figure, plotconfusion(t,y)
% Line Graph
% Train Error Rate
figure; plot(StatisticsNum([1:7],2),StatisticsNum([1:7],3))
hold on
plot(StatisticsNum([8:14],2),StatisticsNum([8:14],3))
plot(StatisticsNum([15:21],2),StatisticsNum([15:21],3))
hold off
title('Train Error Rate'), xlabel('Epochs'),ylabel('Error Rate')
legend('Train Error Rate - 2 Nodes','Train Error Rate - 8 Nodes','Train
Error Rate - 32 Nodes')
% Test Error Rate
figure; plot(StatisticsNum([1:7],2),StatisticsNum([1:7],4))
hold on
plot(StatisticsNum([8:14],2),StatisticsNum([8:14],4))
plot(StatisticsNum([15:21],2),StatisticsNum([15:21],4))
hold off
title('Test Error Rate'), xlabel('Epochs'),ylabel('Error Rate')
legend('Test Error Rate - 2 Nodes','Test Error Rate - 8 Nodes','Test Error
Rate - 32 Nodes')
% Train Standard Deviation
figure; plot(StatisticsNum([1:7],2),StatisticsNum([1:7],5))
hold on
plot(StatisticsNum([8:14],2),StatisticsNum([8:14],5))
plot(StatisticsNum([15:21],2),StatisticsNum([15:21],5))
hold off
title('Train Standard Deviation'), xlabel('Epochs'),ylabel('Standard
Deviation')
legend('Train Standard Deviation - 2 Nodes','Train Standard Deviation - 8
Nodes','Train Standard Deviation - 32 Nodes')
% Test Standard Deviation
figure; plot(StatisticsNum([1:7],2),StatisticsNum([1:7],6))
hold on
plot(StatisticsNum([8:14],2),StatisticsNum([8:14],6))
plot(StatisticsNum([15:21],2),StatisticsNum([15:21],6))
hold off
title('Test Standard Deviation'), xlabel('Epochs'),ylabel('Standard
Deviation')
legend('Test Standard Deviation - 2 Nodes','Test Standard Deviation - 8
Nodes','Test Standard Deviation - 32 Nodes')
```

Exp2)

```
clear;

% Load Breast Cancer Dataset
load cancer_dataset;
% Define Inputs and Outputs
x = cancerInputs;
t = cancerTargets;

% Define the Training Functin
trainFcn = 'trainscg'; % using "trainscg" Train Function
% Define nodes(hidden layers) and epochs to use in analysis
nodes = 8; % Optimal nodes (hidden layers) from exp1)
trainEpochs = 8; % Optimal epochs from exp1)

xLength = length(x);
classifiersNum = 3:2:25;

for classifiersIndexList = 1:length(classifiersNum)
    numNN = 30;
    for i = 1:numNN
        randPermx = randperm(xLength);
        randIndTest = randPermx(1:length(randPermx)/2);
        randIndTrain = randPermx(length(randIndTest)+1 :
length(randPermx));
        trainT = [];
        testT = [];
        trainY = [];
        testY = [];

        for classifiersIndex = 1:classifiersNum(classifiersIndexList)
            % Define the Neural Network
            net = patternnet(nodes, trainFcn);
            net.trainParam.epochs = trainEpochs;

            % Choose Input and Output Pre/Post-Processing Functions
            net.input.processFcns = {'removeconstantrows','mapminmax'};

            %Setup Division of Data for Training, Testing
            net.divideFcn = 'divideind'; % Divide data by Indicies
            net.divideMode = 'sample';  % Divide up every sample
            net.divideParam.trainInd = randIndTrain;
            net.divideParam.testInd = randIndTest;

            % Choose a Performance Function
            net.performFcn = 'crossentropy';  % Cross-Entropy

            % Choose Confusion Plot Function
            net.plotFcns = {'plotconfusion'};

            % Train the net
            [net,tr] = train(net, x, t);

            % Test network
            y = net(x);

            % Change the vectors to index
            tind = vec2ind(t);
```

```
            yind = vec2ind(y);

            trainT = [trainT; tind(tr.trainInd)];
            testT = [testT; tind(tr.testInd)];
            trainY = [trainY; yind(tr.trainInd)];
            testY = [testY; yind(tr.testInd)];
        end
        % Calculate majority with mode function
        ttrainmajority = mode(trainT);
        ttestmajority = mode(testT);
        ytrainmajority = mode(trainY);
        ytestmajority = mode(testY);
        % Calculate Error Rate percentage for Train and Test
        percentErrorsTrain(i) = sum(ttrainmajority ~=
ytrainmajority)/numel(ttrainmajority);
        percentErrorsTest(i) = sum(ttestmajority ~=
ytestmajority)/numel(ttestmajority);
    end
    percentErrorsAverageTrain(classifiersIndexList) =
mean(percentErrorsTrain);
    percentErrorsStdTrain(classifiersIndexList) = std(percentErrorsTrain);

    percentErrorsAverageTest(classifiersIndexList) =
mean(percentErrorsTest);
    percentErrorsStdTest(classifiersIndexList) = std(percentErrorsTest);
end

testErrorGraph = figure;
set(0, 'CurrentFigure', testErrorGraph);
test_legend_str = 'Test Error Rate';
plot(classifiersNum, percentErrorsAverageTest, 'DisplayName',
test_legend_str);
title('Ensembled Classifiers Test Error Rates');
legend show;
xlabel('Number of Classifiers');
ylabel('Error Rate');
```

Exp3)

```
clear;

% Load Breast Cancer Dataset
load cancer_dataset;
% Define Inputs and Outputs
x = cancerInputs;
t = cancerTargets;

% Define the Training Functin
trainFunctions = ["trainscg","trainlm","trainrp"]; % using "trainscg" Train
Function
% Define nodes(hidden layers) and epochs to use in analysis
nodes = 8; % Optimal nodes (hidden layers) from exp1)
trainEpochs = 8; % Optimal epochs from exp1)

xLength = length(x);
classifiersNum = 3:2:25;

for tf = trainFunctions
    trainFcn = tf;
    for classifiersIndexList = 1:length(classifiersNum)
```

```
        numNN = 30;
        for i = 1:numNN
            randPermx = randperm(xLength);
            randIndTest = randPermx(1:length(randPermx)/2);
            randIndTrain = randPermx(length(randIndTest)+1 :
length(randPermx));
            trainT = [];
            testT = [];
            trainY = [];
            testY = [];

            for classifiersIndex = 1:classifiersNum(classifiersIndexList)
                % Define the Neural Network
                net = patternnet(nodes, trainFcn);
                net.trainParam.epochs = trainEpochs;

                % Choose Input and Output Pre/Post-Processing Functions
                net.input.processFcns = {'removeconstantrows','mapminmax'};

                %Setup Division of Data for Training, Testing
                net.divideFcn = 'divideind'; % Divide data by Indicies
                net.divideMode = 'sample';  % Divide up every sample
                net.divideParam.trainInd = randIndTrain;
                net.divideParam.testInd = randIndTest;

                % Choose a Performance Function
                net.performFcn = 'crossentropy';  % Cross-Entropy

                % Choose Confusion Plot Function
                net.plotFcns = {'plotconfusion'};

                % Train the net
                [net,tr] = train(net, x, t);

                % Test network
                y = net(x);

                % Change the vectors to index
                tind = vec2ind(t);
                yind = vec2ind(y);

                trainT = [trainT; tind(tr.trainInd)];
                testT = [testT; tind(tr.testInd)];
                trainY = [trainY; yind(tr.trainInd)];
                testY = [testY; yind(tr.testInd)];
            end
            % Calculate majority with mode function
            ttrainmajority = mode(trainT);
            ttestmajority = mode(testT);
            ytrainmajority = mode(trainY);
            ytestmajority = mode(testY);
            % Calculate Error Rate percentage for Train and Test
            percentErrorsTrain(i) = sum(ttrainmajority ~=
ytrainmajority)/numel(ttrainmajority);
            percentErrorsTest(i) = sum(ttestmajority ~=
ytestmajority)/numel(ttestmajority);
        end
        percentErrorsAverageTrain(classifiersIndexList) =
mean(percentErrorsTrain);
```

```
        percentErrorsStdTrain(classifiersIndexList) =
std(percentErrorsTrain);

        percentErrorsAverageTest(classifiersIndexList) =
mean(percentErrorsTest);
        percentErrorsStdTest(classifiersIndexList) =
std(percentErrorsTest);
    end
    testErrorGraph = figure;
    set(0, 'CurrentFigure', testErrorGraph);
    test_legend_str = 'Test Error Rate';
    plot(classifiersNum, percentErrorsAverageTest, 'DisplayName',
test_legend_str);
    title(sprintf('Ensembled Classifiers Test Error Rates for %s
optimiser',tf));
    legend show;
    xlabel('Number of Classifiers');
    ylabel('Error Rate');
end
```