EECE 6036: Intelligent Systems
Satish Kumar Loganathan

1. Multi-layer feed forward network with configurable network structure (number of layers and neurons), learning rate and momentum to determine whether or not a patient has metabolic disorder, by taking into account the patient's sodium level (L) and blood pressure (P). Patients are labelled as positive (D=1) or negative (D=0) for the disease by the classifier.

Approach: A brief description and justification of your overall strategy (i.e., how chose the parameters that you did, and why).

As in the previous homework, the attribute values were scaled to have unit-variance in order to avoid the formation of a biased model.

Attribute	Range	Variance
Sodium Level	23.0	23.8475576622
Blood Pressure	115.49	242.942427396

Analysis prior to Scaling (Pre-processing)

The below table shows the spread of the attribute values after Scaling.

Attribute	Range	Variance
Sodium Level	4.70983705521	1.0
Blood Pressure	7.40956475926	1.0

Analysis after Scaling (Pre-processing)

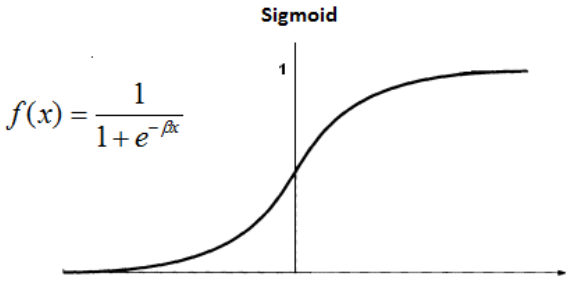
The configurable multi-layer feed-forward neural network was implemented in Python, with the following configurable parameters:

i. Number of hidden layers
ii. Number of hidden neuron in each hidden layer
iii. Number of neuron in the output layer
iv. Learning rate, which can be vary for each layer
v. Momentum

The initial **weights** and **bias** of the feed forward neural network are chosen as very small random values ranging from 0 to 0.01.

Since the model needs to handle a 0 or 1 classification only, the number of **output neurons** was set to 1. The neuron is expected to fire when the patient has metabolic order and lie dormant otherwise.

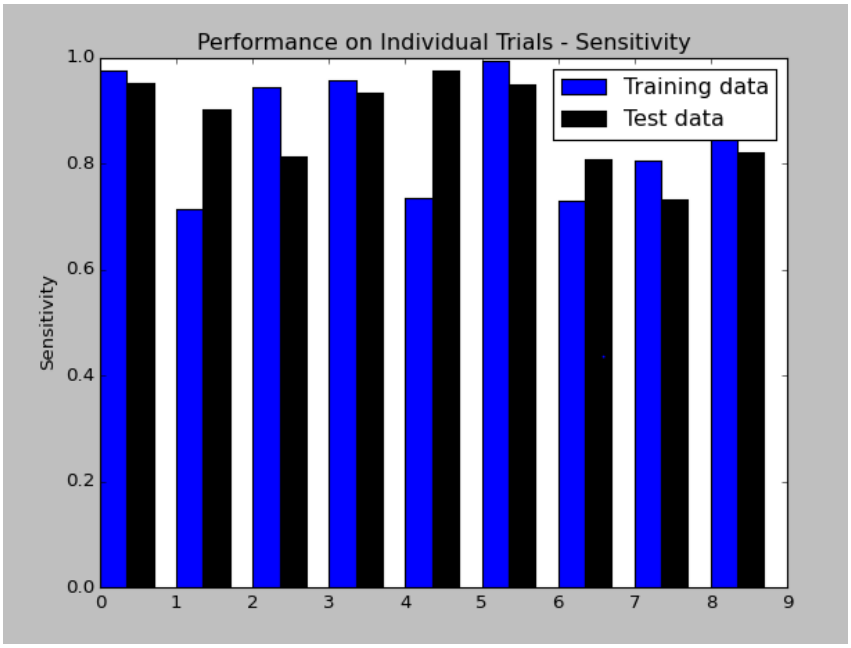
The model utilizes a **sigmoid activation function**, which yields results almost similar to the step function. Activation thresholds of **0.80** and **0.25** were adopted for discretization, meaning that any value above 0.85 would be considered as 1 and any value below 0.25 would be considered as 0. Once the forward-pass is completed and the data point is classified, the **Back Propagation** algorithm is utilized to update the weights and this in turn makes use of the derivate of the sigmoid function.



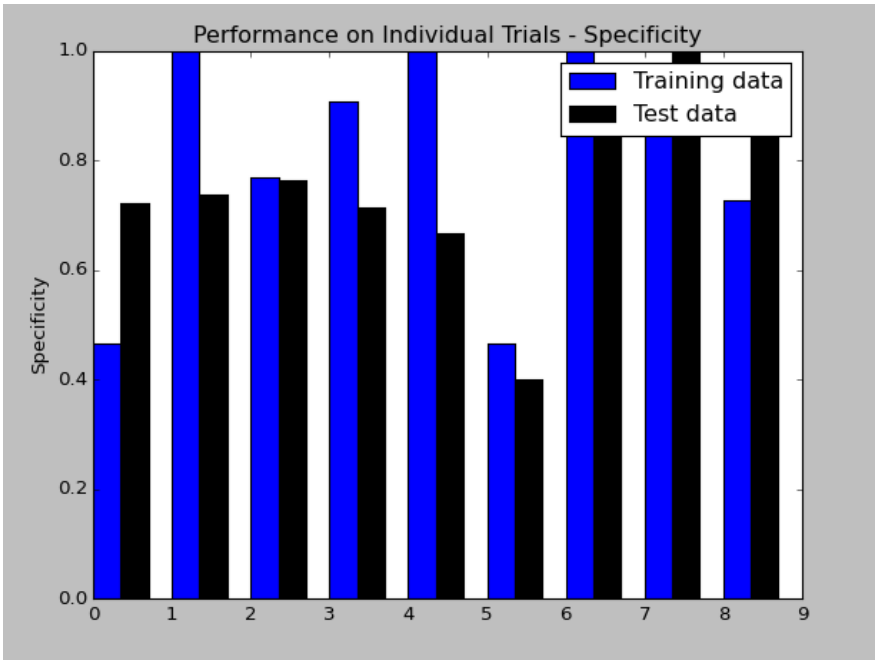
The network structure, learning rate and momentum were determined by trial and error method. The best model attained utilizes a single hidden layer with **25** hidden neurons, a learning rate of **0.05** and a momentum of **0.1**. The model was trained over **100** epochs. The number of epochs was chosen with an intention to be able to study the convergence of the perceptron over a prolonged training period. As instructed, 9 trials were performed across both the algorithms and randomly ordered input data. The random ordering lies along the same lines as k-fold cross validation where it is ensured that the final outcome i.e. the results are not biased by the selection of the training and test data.

Performance on Individual Trials:

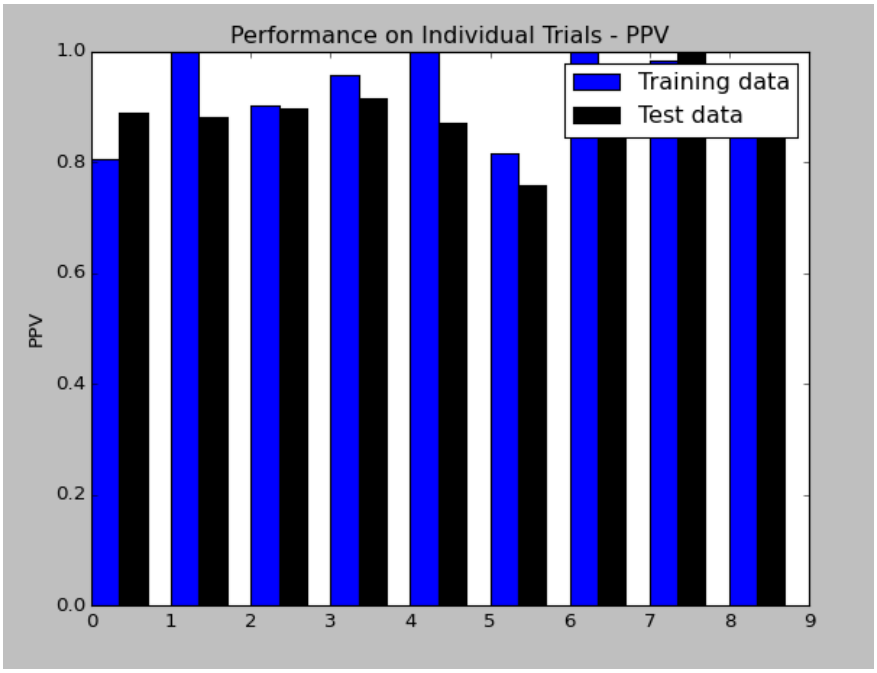
The performance of the individual trials have been depicted in terms of four bar plots, one for each of the evaluation measures – sensitivity, specificity, Positive Predicted Value (PPV) and Negative Predicted Value (NPV).



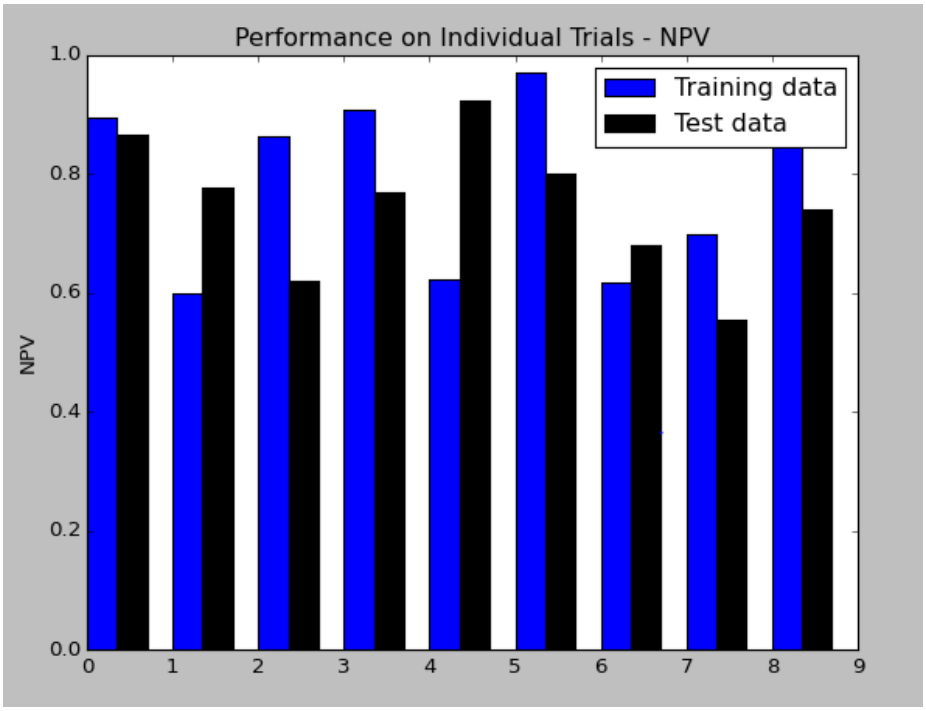
Performance of Individual Trials – Sensitivity



Performance of Individual Trials – Specificity



Performance of Individual Trials – Positive Predicted Value



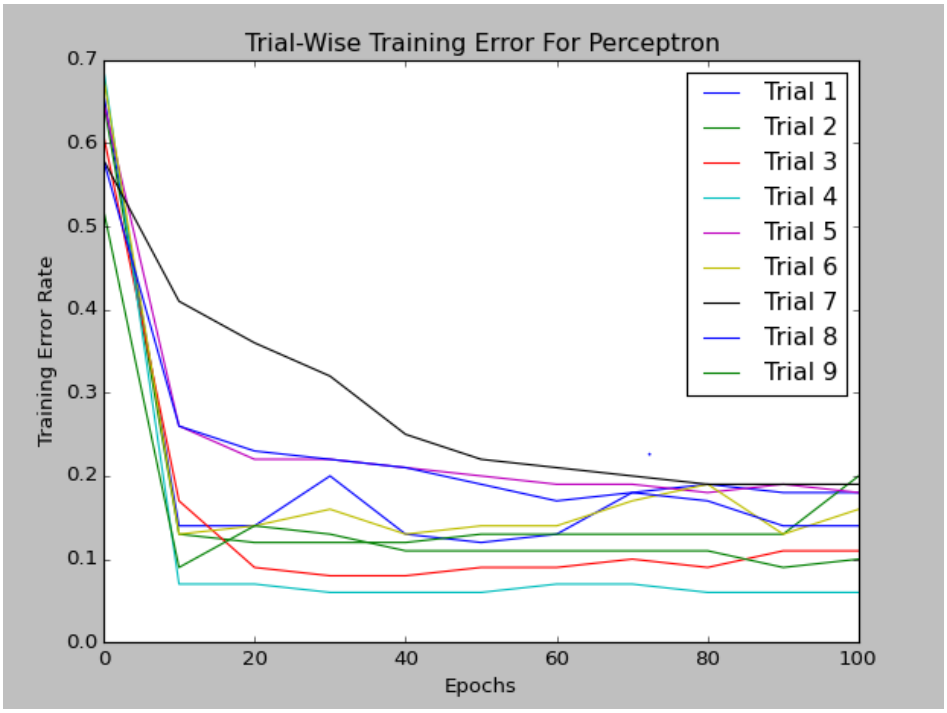
Performance of Individual Trials – Negative Predicted Value

Average Performance:

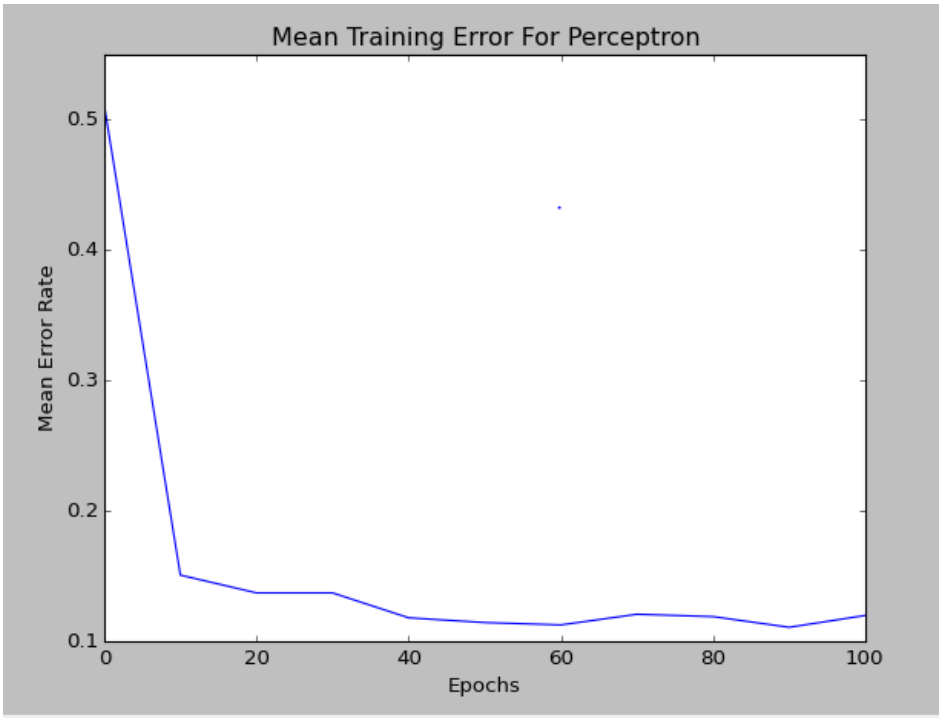
Mean values of sensitivity, specificity, PPV and NPV for both training and testing data averaged over all 9 trials, along with the standard deviation for each case.

Evaluation Metrics	Training Set	Test Set
Sensitivity	0.8699 ± 0.011	0.8769 ± 0.079
Specificity	0.8124 ± 0.208	0.7668 ± 0.173
Predicted Positive Value	0.9292 ± 0.073	0.9061 ± 0.067
Predicted Negative Value	0.7877 ± 0.141	0.7479 ± 0.109

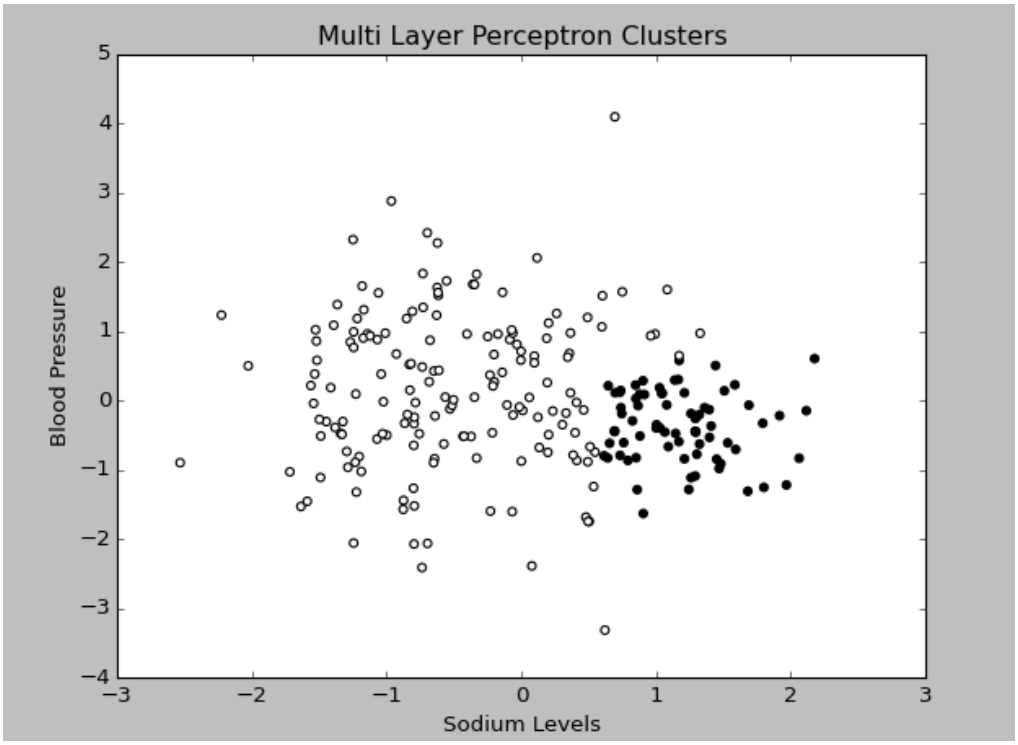
Trial-Wise Training Error:



Mean Training Error:



Perceptron Decision Boundary – Cluster formations:



Analysis of Results:

It can be seen that the multi-layer feed forward neural network has performed exceeding well in terms of specificity, sensitivity, PPV and NPV even with **linearly inseparable** data. The mean training error and trail-wise training error graphs indicate that the convergence has been steep between 10 and 20 epochs and the following epochs did not have a significant impact on improving the performance of the model. This could also be majorly attributed to the initial scaling of the data points.

An important aspect to consider while choosing a classifier would be the nature of the classification problem. It has been stated that the medication for the disease can have adverse side-effects and hence, it is crucial to ensure that a person who does not have metabolic disorders is not diagnosed positively. Thus, the measure of interest would be sensitivity which analyzes the fraction of negative cases that were correctly produced, i.e. the number of patients who were rightly excluded as not having metabolic disorders.

The multi-layer feed forward network exhibited sensitivity values of over 0.85 on both the training and the test data set and hence, servers to be a very good classifier for the given problem.

---

2. Multi-layer feed forward network to classify the MNIST dataset, which provides images of hand written characters and digits. In this implementations, only the numbers subset is being considered for this problem.

System Description:

The python modules implemented in the previous problem were tweaked and utilized to classify hand written digits from the MNIST dataset. However, with 5000 records and each record containing 784 attributes (each pertaining to a pixel value), training the multi-layer feed forward network over and over again with varied network parameters, in order to identify the best suited model was a challenging task.

Hence, the strategy that I adopted was to utilize a random sample of 500 records from the MNIST data set for initially training the model and choosing the network parameters. The data is randomly

split into training and test sets with an **80:20** split. The initial number of **epochs** was set to 80 to be able to analyze the performance of the network over the sample and determine the appropriate number.

As in the previous problem, the **activation function** utilized was the sigmoid function with an upper threshold of 0.80 and a lower threshold of 0.25. The number of **output neurons** was set to 10, to match the number of hand written digits available. The network was trained in such a manner that each neuron in the output layer detects a particular digit. For example, when the digit to be recognized is 2, only the third (0, 1, 2) output neuron fires and all other output neurons remain dormant. The output layer is designed to follow a **Soft Max** approach when classifying the test data, where the neurons with the output is considered as the digit corresponding to the neuron with the highest activation.

Also, for the system to be able to cope up with the volume of the data in the MNIST data set, a **stochastic gradient descent learning** approach was adopted for training the network. At every epoch, a random sub-sample containing 40% of the training data was presented to the multi-layer feed forward network and the network was trained and validated over the same.

Choice of network parameters:
As stated earlier, to initially determine the network parameters, the model was repeatedly trained and validated against a representative sample of the entire dataset. The initial number of epochs was set to 80.

- The number of hidden neurons was varied from 50 to 150 with manual increments of 25 and the network was observed to produce the best results starting with 100 hidden neurons. Adding more neurons only resulted in slowing down the system and did not produce any significant improvements over the error rate.
- Similarly, the learning rate and momentum were manually varied and happened to provide maximum impact at **0.5** and **0.1** respectively. These values also determine the number of epochs since, all the three are inter dependent.
- The ideal number of epochs was observed to be **40**, beyond which the model was being over-fitted and the error rate started rising again.

The time series of the training error rate, capturing the initial training error and the training error at the end of every 10^{th} epoch, can be observed in Figure 2.1. Similar results across the entire data set is shown in Figure 2.2.

Results:

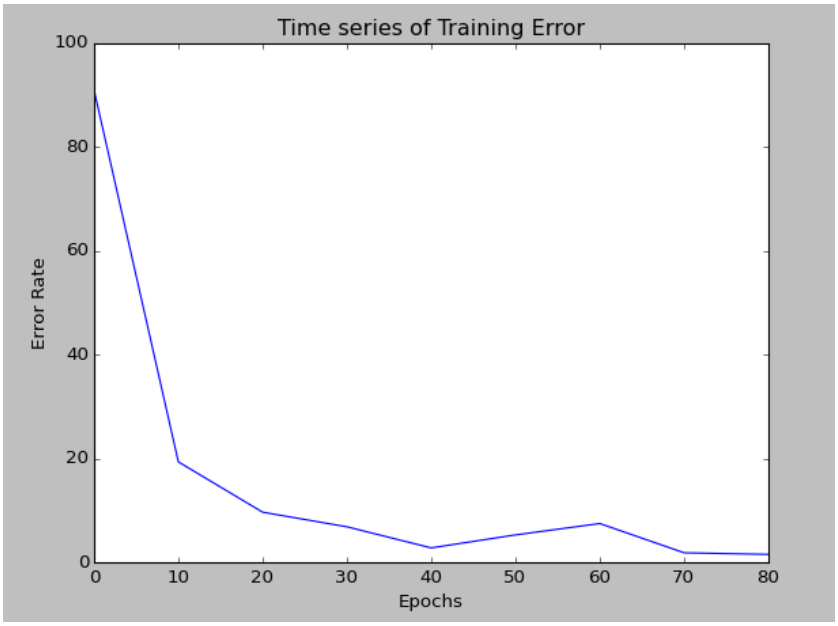


Fig 2.1 Time series of Training Error against sampled data

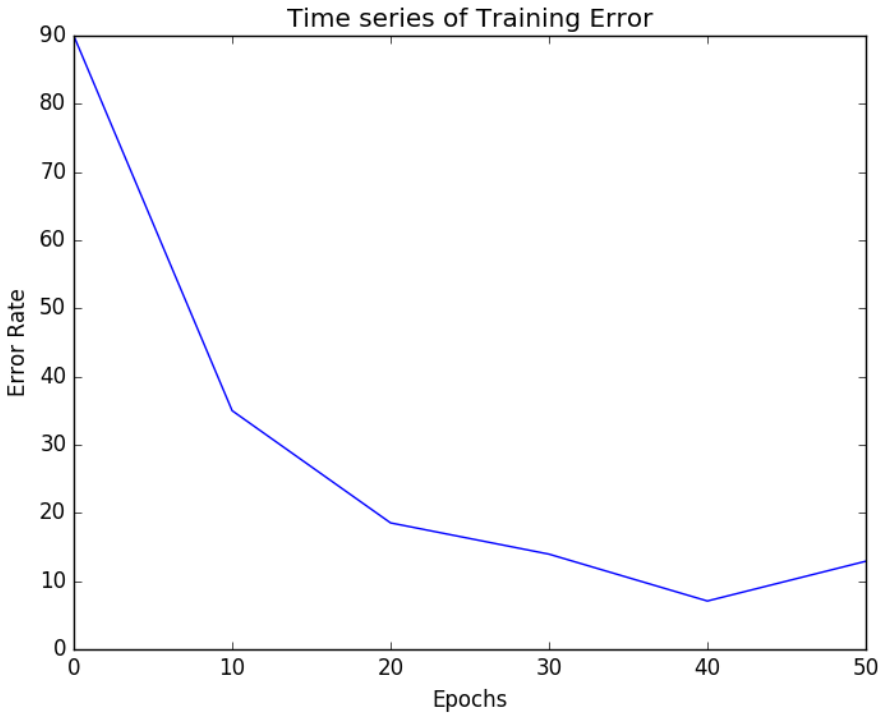


Fig 2.2 Time series of Training Error against the entire dataset

Misclassified data points:

Sample data

Training Set	3.33%
Test Set	16.45%

Entire dataset

Training Set	13.85%
Test Set	19.16%

Analysis of Results:

In terms of network performance, the training and test error observed with the sampled data are fairly dissimilar. This is owing to the fact that the number of data points presented to the model for training at every epoch were not sufficient to build a model that was mature enough to classify the hand-written digits in the MNIST dataset.

However, the results attained with the entire data set vary significantly less in comparison. Also, it can be observed that the training error was minimum at the 40^{th} epoch with the sampled data as well as the entire data set, and hence, this was selected as the ideal point to stop training the model. Further iterations would only result in an over-fitted model that becomes inefficient with classifying novel data points.

In terms of the data, the hand-written characters are represented with 18x18 pixels, with several pixel values remaining the same across a subset of the digits (eight and nine for example). This indicates high degrees overlapping or the presence of closely spaced clusters. Also, if we consider the problem of isolating all data points to corresponding to a single hand-written digit from the rest, this would be a linearly inseparable problem. This explains why the network required over 100 hidden neurons to be able to efficiently classify the hand-written digits provided in the MNIST dataset.

The model exhibited an accuracy of over 77% with novel patterns in the test data, demonstrating the predictive capabilities of multi-layer feed forward networks.

Appendix: Programs:

Home.py

```
from sklearn import preprocessing

import numpy as np
import Constants as cts
import MLP_Backpropagation as mlp
import PlotBarGraph
import ComputeMean
import PlotTrainingError

dataSet = np.genfromtxt('hw2_dataProblem.txt', skip_header=2)
num_trials = 9

# Analyzing the spread of data to determine a need for data pre-processing.
print 'Evaluation prior to scaling:'
print 'Range - Sodium level', np.ptp(dataSet[:, cts.SODIUM_LEVEL])
print 'Range - Blood pressure', np.ptp(dataSet[:, cts.BLOOD_PRESSURE])
print 'Variance - Sodium Level', np.var(dataSet[:, cts.SODIUM_LEVEL])
print 'Variance - Blood pressure', np.var(dataSet[:, cts.BLOOD_PRESSURE])

dataSet[:, [cts.SODIUM_LEVEL, cts.BLOOD_PRESSURE]] = \
    preprocessing.scale(dataSet[:, [cts.SODIUM_LEVEL, cts.BLOOD_PRESSURE]])

print '\n Evaluation after scaling:'
print 'Range - Sodium level', np.ptp(dataSet[:, cts.SODIUM_LEVEL])
print 'Range - Blood pressure', np.ptp(dataSet[:, cts.BLOOD_PRESSURE])
print 'Variance - Sodium Level', np.var(dataSet[:, cts.SODIUM_LEVEL])
print 'Variance - Blood pressure', np.var(dataSet[:, cts.BLOOD_PRESSURE])

perceptron_training_results = []
perceptron_test_results = []
perceptron_training_error = []

num_hidden_layers = 1
num_hidden_neurons = [25]
num_output_neurons = 1
learning_rate = 0.05
momentum = 0.1
plot_decision_boundary = False

for idx in range(num_trials):
    # shuffling the items in the data set
    print "Trial ", idx+1
    np.random.shuffle(dataSet)
    if idx == num_trials-1:
        plot_decision_boundary = True
    result, training_error = mlp.train_and_classify(dataSet, num_hidden_layers,
num_hidden_neurons, num_output_neurons, learning_rate, momentum, True,
plot_decision_boundary)
    perceptron_training_results.append(result)
    perceptron_training_error.append(training_error)
```

```
    plot_decision_boundary = False
    perceptron_test_results.append(mlp.train_and_classify(dataSet,
num_hidden_layers, num_hidden_neurons,
num_output_neurons, learning_rate, momentum,False, plot_decision_boundary))

print 'Perceptron - Training:\n', perceptron_training_results
print 'Perceptron - Test:\n', perceptron_test_results

PlotBarGraph.plot_bar_graphs(perceptron_training_results,
perceptron_test_results, num_trials)

print '\nPerceptron training results:'
ComputeMean.compute_mean(perceptron_training_results)
print '\nPerceptron test results:'
ComputeMean.compute_mean(perceptron_test_results)

PlotTrainingError.plot_training_error(perceptron_training_error, num_trials)
```

MLP_Backpropagation.py:

```
from __future__ import division
import numpy as np
import random as rand
import Constants as cts
import math
import EvaluateClassifier as eval
import matplotlib.pyplot as plt


activation_threshold = 0.80


# Sigmoid activation function
def sigmoid_activation(x):
    if (1 / (1 + math.exp(-x))) > activation_threshold:
        return 1
    else:
        return 0


# Function that returns the gradient i.e. derivative of sigmoid(x)
def gradient_sigmoid(x):
    exp = math.exp(-x)
    return exp / ((1 + exp) ** 2)


# Function to train a multi-layer perceptron and validate the model using cross
validation
#       2D array  data                data set with attributes and class
labels
#       Integer   num_hidden_layers    number of hidden layers
#       1D array  num_hidden_layers    1D array containing the number of
neurons in each hidden layer
#       Integer   num_output_neurons   number of output neurons
```

```
#       Float     learning_rate
#       Float     momentum
def train_and_classify(data, num_hidden_layers, num_hidden_neurons,
num_output_neurons, learning_rate, momentum,
                       is_training_set, plot_decision_boundary):
    num_epochs = 100
    training_error = []

    # Partitioning training and test data
    num_records = np.size(data, 0)
    training_size = int(np.floor(0.8 * num_records))

    training_data = data[0:training_size, :]
    test_data = data[training_size:num_records, :]

    # initialize weights - random values
    num_neurons = []
    num_neurons[:] = num_hidden_neurons[:]
    num_neurons.append(num_output_neurons)
    weights = {}
    changes = {}

    print "Initializing weights..."
    for layer in range(num_hidden_layers+1):
        weight_matrix = []
        changes_matrix = []
        for neuron_idx in range(num_neurons[layer]):
            row = []
            change = []
            if layer == 0:  # Initial layer will receive direct input signals
                for idx in range(len(data[0])):
                    row.append(rand.random())
                    change.append(0)

            else:  # The other layers will receive signals from the previous
layer
                for idx in range(num_neurons[layer-1]+1):
                    row.append(rand.random())
                    change.append(0)
            weight_matrix.append(row)
            changes_matrix.append(change)
        weights[layer] = weight_matrix
        changes[layer] = changes_matrix

    # Initial error prior to training
    training_error.append(classify(training_data, num_hidden_layers,
num_neurons, weights, True, False))

    delta = {}
    activation = {}
    summation = {}

    print "Training the multi-layer perceptron... "
    for epoch in range(num_epochs):
        for data_idx in range(training_size):
```

```
            data_pt = training_data[data_idx]
            # Iterating through the data points and computing the output of
each layer
            for layer in range(num_hidden_layers + 1):
                weight_matrix = weights[layer]
                activation_results = []
                summation_results = []
                for neuron_idx in range(num_neurons[layer]):
                    weighted_sum = 0
                    if layer == 0:  # Initial layer
                        # Summation -> weights * input + bias
                        for idx in range(len(data_pt)-1):
                            weighted_sum += weight_matrix[neuron_idx][idx] *
data_pt[idx]
                    else:
                        # Summation -> weights * input + bias
                        for idx in range(num_neurons[layer - 1]):
                            weighted_sum += weight_matrix[neuron_idx][idx] *
activation[layer-1][idx]
                    weighted_sum += weight_matrix[neuron_idx][idx+1]  # bias
                    # Activation function -> Sigmoid
                    summation_results.append(weighted_sum)
                    activation_results.append(sigmoid_activation(weighted_sum))
                activation[layer] = activation_results
                summation[layer] = summation_results

            # Updating the weights - Back Propagation
            for layer in range(num_hidden_layers, -1, -1):
                is_output_layer = False
                activation_results = activation[layer]
                summation_results = summation[layer]
                delta_values = []
                if layer == num_hidden_layers:
                    is_output_layer = True
                else:
                    outgoing_weight_matrix = weights[layer+1]
                for neuron_idx in range(num_neurons[layer]):
                    delta_temp = 0
                    if is_output_layer:
                        delta_temp += (data_pt[cts.CLASS_LABEL] -
                                       activation_results[neuron_idx]) *
gradient_sigmoid(summation_results[neuron_idx])
                    else:
                        for idx in range(num_neurons[layer+1]):
                            delta_temp +=
outgoing_weight_matrix[idx][neuron_idx]*delta[layer+1][idx]
                        delta_temp *=
gradient_sigmoid(summation_results[neuron_idx])
                    delta_values.append(delta_temp)
                delta[layer] = delta_values

            for layer in range(num_hidden_layers, -1, -1):
                delta_values = delta[layer]
                if layer is not 0:
                    prev_activation_results = activation[layer-1]
```

```
                for neuron_idx in range(num_neurons[layer]):
                    # Update weights when delta is not zero
                    delta_value = delta_values[neuron_idx]
                    if delta_value is not 0:
                        if layer == 0:  # Initial layer
                            for idx in range(len(data_pt)-1):
                                change = learning_rate*delta_value*data_pt[idx]
+ \

momentum*changes[layer][neuron_idx][idx]
                                weights[layer][neuron_idx][idx] += change
                                changes[layer][neuron_idx][idx] = change
                        else:  # The other layers
                            for idx in range(num_neurons[layer-1]):
                                change =
learning_rate*delta_value*prev_activation_results[idx] + \

momentum*changes[layer][neuron_idx][idx]
                                weights[layer][neuron_idx][idx] += change
                                changes[layer][neuron_idx][idx] = change
                        change = learning_rate*delta_value +
momentum*changes[layer][neuron_idx][idx+1]
                        weights[layer][neuron_idx][idx+1] += change  # Updating
bias
                        changes[layer][neuron_idx][idx+1] = change
        if (epoch + 1) % 10 == 0 and is_training_set:
            training_error.append(classify(training_data, num_hidden_layers,
num_neurons, weights, True, False))

    print "Training complete..."
    print "Classifying test data and obtaining results..."
    if is_training_set:
        return classify(training_data, num_hidden_layers, num_neurons, weights,
False,
                        plot_decision_boundary), training_error
    else:
        return classify(test_data, num_hidden_layers, num_neurons, weights,
False,
                        plot_decision_boundary)


# Function to classify the data points and evaluate the model.
def classify(data, num_hidden_layers, num_neurons, weights,
only_training_error, plot_decision_boundary):
    true_positive = 0.001
    false_positive = 0.001
    true_negative = 0.001
    false_negative = 0.001
    predicted_class_labels = []

    activation = {}
    for data_idx in range(len(data)):
        data_pt = data[data_idx]
        # Iterating through the data points and computing the output of each
layer
```

```
        for layer in range(num_hidden_layers + 1):
            weight_matrix = weights[layer]
            activation_results = []
            for neuron_idx in range(num_neurons[layer]):
                weighted_sum = 0
                if layer == 0:  # Initial layer
                    # Summation -> weights * input + bias
                    for idx in range(len(data_pt)-1):
                        weighted_sum += weight_matrix[neuron_idx][idx] *
data_pt[idx]
                else:
                    # Summation -> weights * input + bias
                    for idx in range(num_neurons[layer - 1]):
                        weighted_sum += weight_matrix[neuron_idx][idx] *
activation[layer-1][idx]
                weighted_sum += weight_matrix[neuron_idx][idx+1]  # bias
                # Activation function -> Sigmoid
                activation_results.append(sigmoid_activation(weighted_sum))
            activation[layer] = activation_results

        predicted_class = activation_results[0]
        predicted_class_labels.append(predicted_class)

        if predicted_class == data_pt[cts.CLASS_LABEL]:
            if predicted_class > 0:
                true_positive += 1
            else:
                true_negative += 1
        else:

            if predicted_class > 0:
                false_positive += 1
            else:
                false_negative += 1
    if plot_decision_boundary:
        plot_clusters(data, predicted_class_labels)
    if only_training_error:
        incorrect_classifications = false_negative + false_positive
        return format(incorrect_classifications/len(data), '.2f')
    else:
        return eval.evaluate_classifier(true_positive, true_negative,
false_positive, false_negative)


# Function to plot the decision boundary formed by the multi-layer perceptron
def plot_clusters(data_set, class_labels):
    plot_data = np.delete(data_set, [cts.CLASS_LABEL], axis=1)
    plt.scatter(plot_data[:, cts.SODIUM_LEVEL], plot_data[:,
cts.BLOOD_PRESSURE], c=class_labels, cmap='gray')
    plt.xlabel('Sodium Levels')
    plt.ylabel('Blood Pressure')
    plt.title('Multi Layer Perceptron Clusters')
    plt.show()
```

EvaluateClassifier.py:

```
# Function to compute and display the sensitivity, specificity, PPV and NPV of
a classifier.
from __future__ import division


def evaluate_classifier(true_positive, true_negative, false_positive,
false_negative):

    sensitivity = true_positive / (true_positive + false_negative)
    specificity = true_negative / (false_positive + true_negative)
    PPV = true_positive / (true_positive + false_positive)
    NPV = true_negative / (true_negative + false_negative)

    return [sensitivity, specificity, PPV, NPV]
```

PlotTrainingError.py:

```
import matplotlib.pyplot as plt
import numpy as np


def plot_training_error(training_error, num_trails):
    # Function to plot the training error and the mean training error
associated with the Perceptron.

    x_values = np.array(range(0, 110, 10))
    mean_training_error = training_error[0]
    for idx1 in range(len(training_error[0])):
        mean_training_error[idx1] = float(training_error[0][idx1])
    plt.figure(1)
    for idx in range(num_trails):
        plt.plot(x_values, np.array(training_error[idx]))
        if idx > 0:
            for idx1 in range(len(training_error[idx])):
                mean_training_error[idx1] += float(training_error[idx][idx1])
        plt.autoscale(enable=True, axis=u'both', tight=False)
    plt.xlabel('Epochs')
    plt.ylabel('Training Error Rate')
    plt.title('Trial-Wise Training Error For Perceptron')
    plt.legend(['Trial 1', 'Trial 2', 'Trial 3', 'Trial 4', 'Trial 5', 'Trial
6', 'Trial 7', 'Trial 8', 'Trial 9'])

    size = len(mean_training_error)
    for idx1 in range(size):
        mean_training_error[idx1] /= size

    plt.figure(2)
    plt.plot(x_values, mean_training_error)
    plt.xlabel('Epochs')
    plt.ylabel('Mean Error Rate')
    plt.title('Mean Training Error For Perceptron')
```

```
    plt.show()
```

PlotBarGraph.py

```
import numpy as np
import matplotlib.pyplot as plt


def plot_bar_graphs(training_results, test_results, num_trials):

    ind = np.arange(num_trials)
    width = 0.35

    labels = ['Sensitivity', 'Specificity', 'PPV', 'NPV']
    for idx1 in range(len(training_results[0])):
        training = []
        test = []
        for idx2 in range(num_trials):
            training.append(training_results[idx2][idx1])
            test.append(test_results[idx2][idx1])

        fig, ax = plt.subplots()
        rects1 = ax.bar(ind, training, width, color='b')

        rects2 = ax.bar(ind + width, test, width, color='k')

        ax.set_ylabel(labels[idx1])
        ax.set_title('Performance on Individual Trials - '+labels[idx1])
        ax.legend((rects1[0], rects2[0]), ('Training data', 'Test data'))

        plt.show()
```

Constants,py:

```
# Constants to define the indices of the dataset.
SODIUM_LEVEL = 0
BLOOD_PRESSURE = 1
CLASS_LABEL = 2
BIAS = 2
```