Assignment 2, COMP4702

Roy Portas, 43560846

May 4, 2017

Question 4.2

```matlab
1   % Q2
2
3   a = randn(200, 2);
4   b = a + 4;
5   c = a;
6
7   c(:, 1) = 3 * c(:, 1);
8   c = c - 4;
9
10  d = [a; b];
11  e = [a; b; c];
12
13  % hold on;
14  % plot(a(:, 1), a(:, 2), '+');
15  % plot(b(:, 1), b(:, 2), 'o');
16  % plot(c(:, 1), c(:, 2), '*');
17
18  % Use the first dataset
19  data = d;
20
21
22  figure;
23  subplot(3, 2, 1);
24  hold on;
25  % ksdensity(data, 'PlotFcn', 'contour');
26  plot(a(:, 1), a(:, 2), '+');
27  plot(b(:, 1), b(:, 2), 'o');
28  plot(c(:, 1), c(:, 2), '*');
29
30  title('Contours Overlay');
31
32  % Calculate the grid
33  [f, xi] = ksdensity(data); x = linspace(min(xi(:, 1)),max(xi(:, 1)
       ));
34  y = linspace(min(xi(:, 2)),max(xi(:, 2)));
35  [xq,yq] = meshgrid(x,y);
36  z = griddata(xi(:, 1),xi(:, 2),f,xq,yq);
37
38  % We now have x, y, z that can be used to get the gradient at any
       point
39
40  contour(x, y, z);
```

```matlab
41  xlim([-10, 10]);
42  ylim([-10, 10]);
43  hold off;
44
45  copy = data;
46  new_points = copy;
47
48
49  lambdas = [1.8, 3, 5];
50  for j = 1:3
51
52      for i = 1:5
53          new_points = step(new_points, x, y, z, lambdas(j));
54          % ksdensity(data, 'PlotFcn', 'contour');
55      end
56      subplot(1, 3, j);
57      hold on;
58      % Plot the data
59      plot(a(:, 1), a(:, 2), '+');
60      plot(b(:, 1), b(:, 2), 'o');
61      %plot(c(:, 1), c(:, 2), '*');
62
63      scatter(new_points(:, 1), new_points(:, 2), 'k', 'LineWidth',
               3);
64      title(sprintf('Lambda %d', lambdas(j)));
65      xlim([-10, 10]);
66      ylim([-10, 10]);
67      hold off;
68  end
69
70
71  function dist = euclid_distance(point1, point2)
72      dist = sqrt((point1(1) - point2(1))^2 + (point1(2) - point2(2)
             )^2);
73  end
74
75  function new_points = step(points, x, y, z, max_distance)
76      % Calibration factor (lambda)
77      % max_distance = 1.8;
78
79      new_points = zeros(length(points), 2);
80      for i = 1:length(points)
81          point = points(i, :);
82
83          numerator = 0;
```

```matlab
84            denominator = 0;
85
86            for j = 1:length(points)
87                other_point = points(j, :);
88                if euclid_distance(point, other_point) < max_distance
89
90                    % Calculate part of sum
91                    distance = euclid_distance(point, other_point);
92                    numerator = numerator + (distance * other_point);
93                    denominator = denominator + distance;
94                end
95            end
96
97            mx = numerator / denominator;
98            new_points(i, :) = mx(1, :);
99        end
100 end
```

Question 4.3

The values 1.8, 3, and 5 were chosen for the lambda values, the results are outlined below.
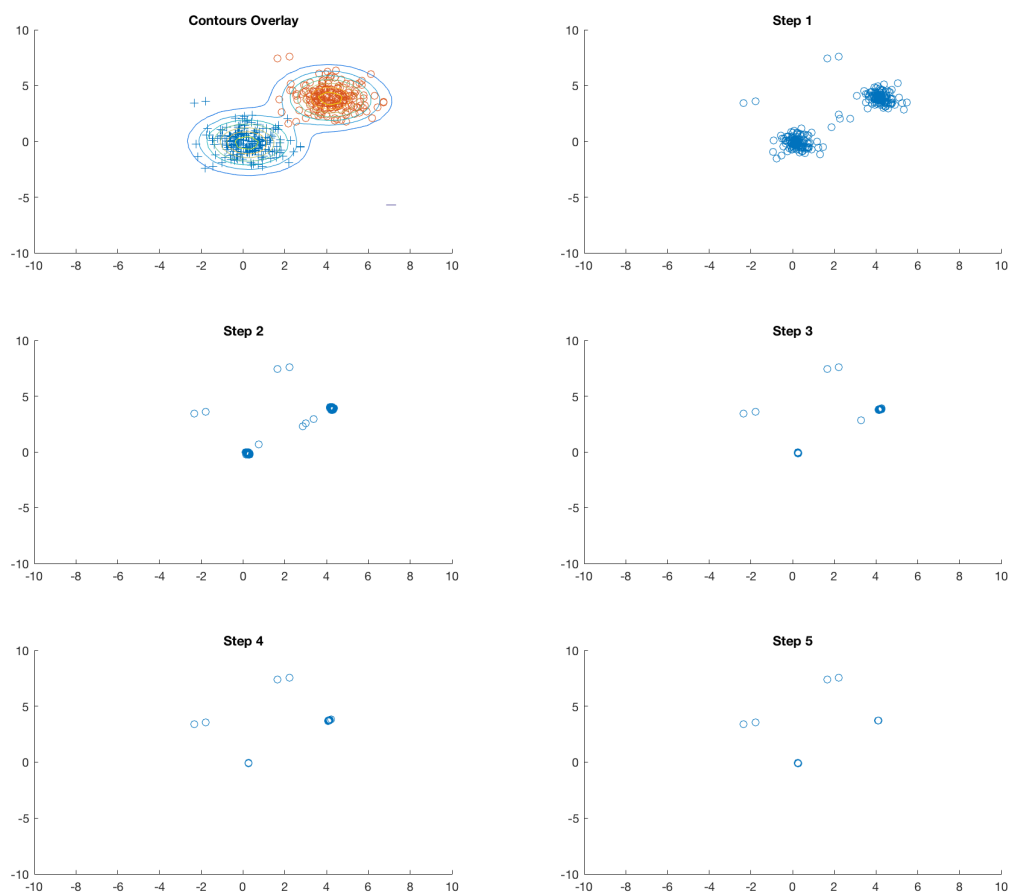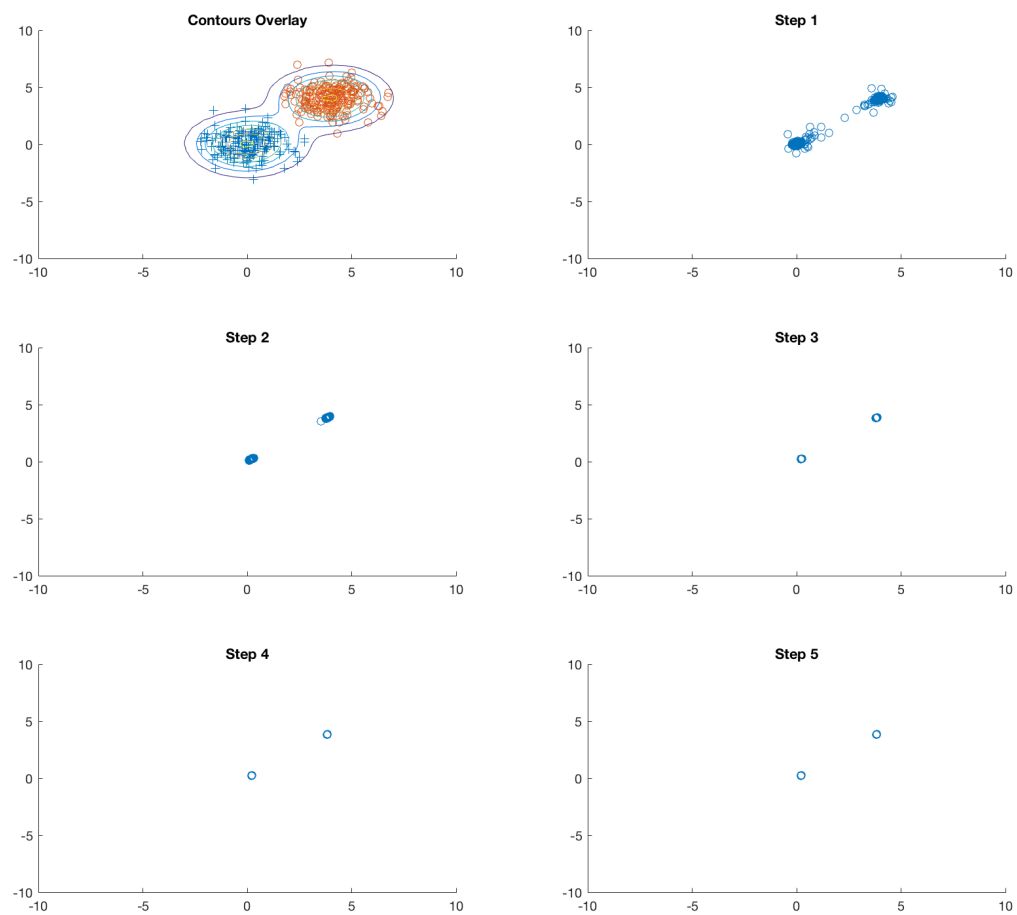
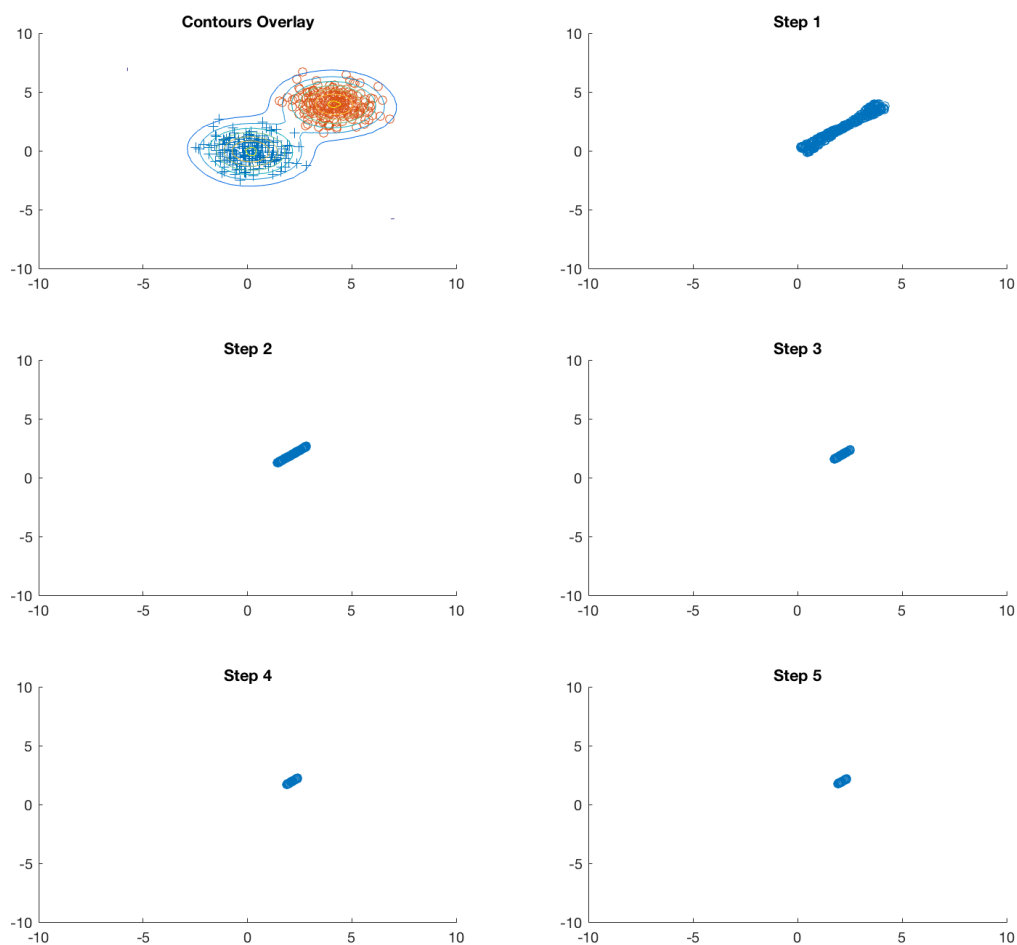Figure 1: 2 Classes, Lambda = 1.8

Figure 2: 2 Classes, Lambda = 3
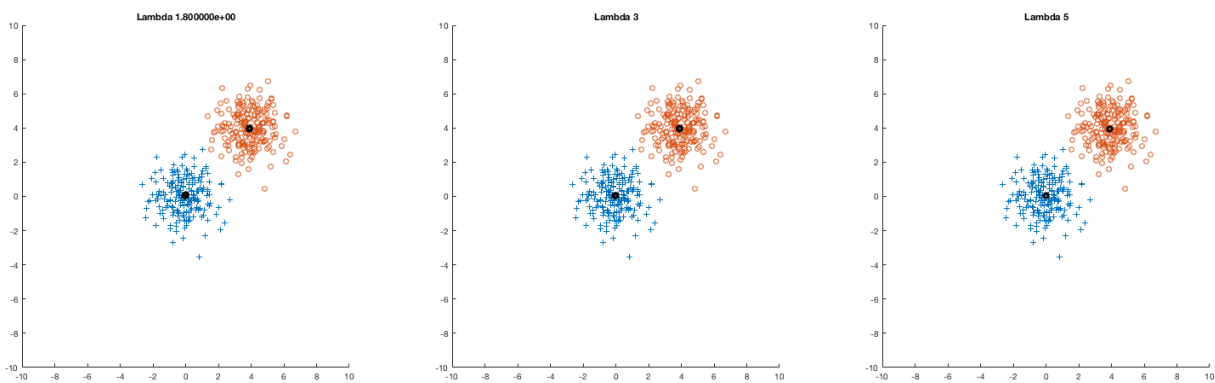
Figure 3: 2 Classes, Lambda = 5

Figure 4: Cluster centres over data

For the 2 class problem, a lambda value set to 3 provided the best result. A lambda of 1.8 cause a few outliers not to shift towards the mean value. Whereas a lambda value of 5 caused all the points to shift towards one of the means, which is not desired. A lambda value of 3 shifted all the points to the two mean values without leaving outliers, thus is the best lambda value out of the three.
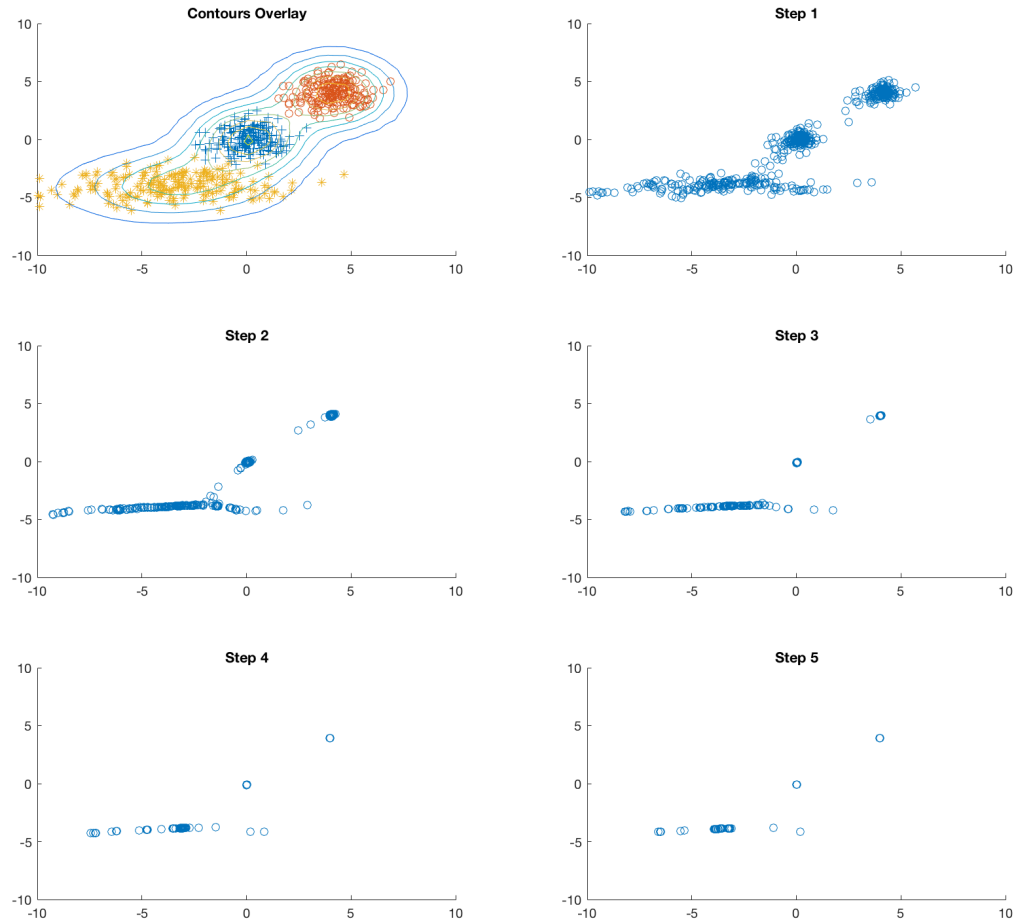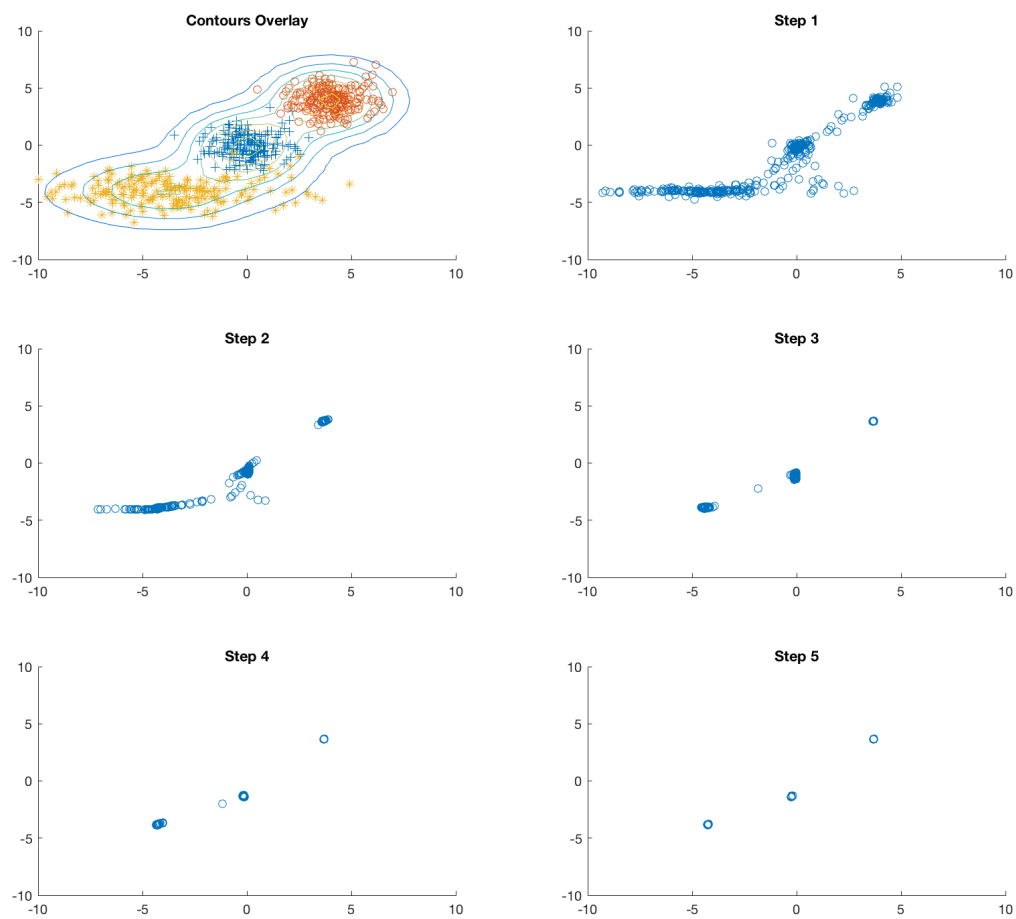

Figure 5: 3 Classes, Lambda = 1.8
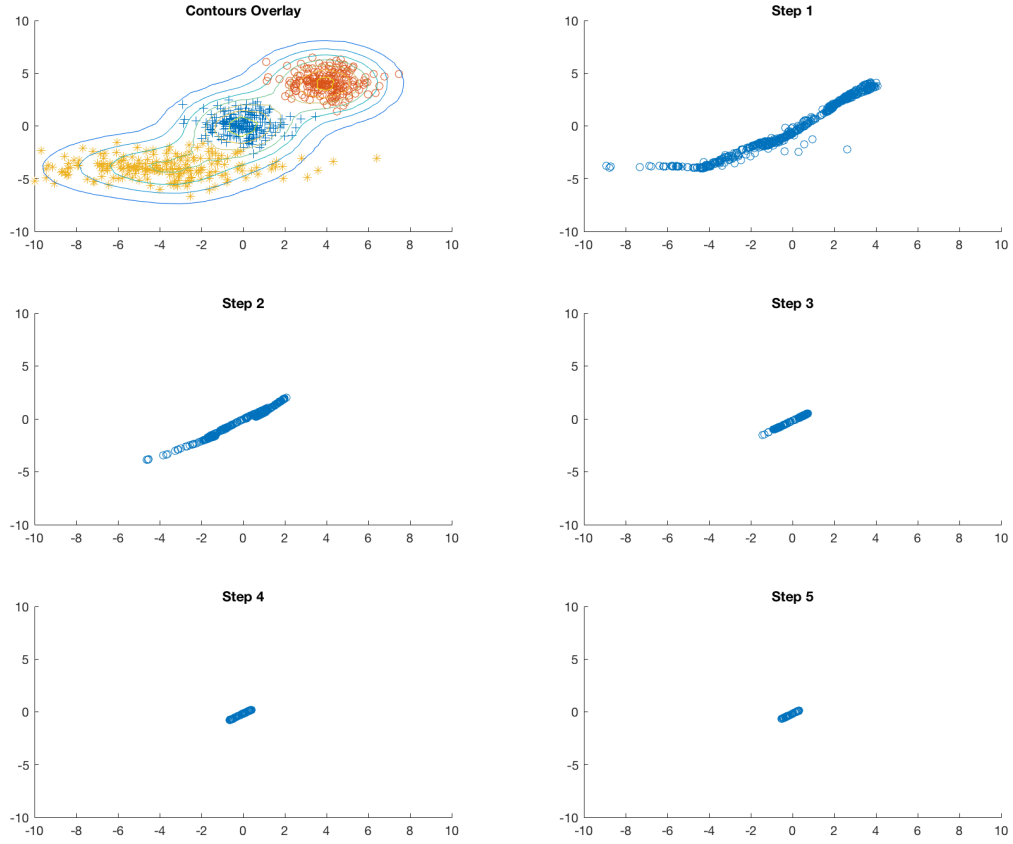
Figure 6: 3 Classes, Lambda = 3

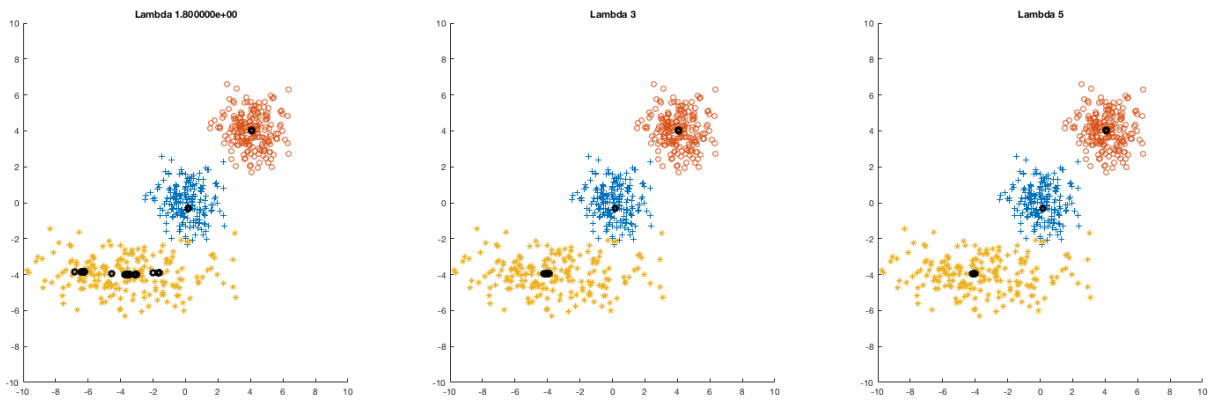Figure 7: 3 Classes, Lambda = 5

Figure 8: Cluster centres over data

The 3 class problem showed similar results with the same lambda values 1.8, 3 and 5. Again the 1.8 lambda value failed to shift some outliers towards the mean and the 5 lambda

9

value shifted all of them towards a single point. The lambda value 3 correctly shifted all the points to their respective means, thus the lambda value of 3 was the best choice out of the three numbers.

Question 5.1

```matlab
1  %
2  % Principle Component Analysis
3  %
4  function result = pca(data)
5      % Find the mean of the data
6      m = mean(data);
7      % Find the covariance on the normalized data
8      S = cov(data - m);
9      % Calculate the eigenvalues and eigenvectors of the covariance
            matrix
10     [evec, eval] = eig(S);
11
12     % Sort the eigenvalues
13     [y, i] = sort(diag(eval), 'descend');
14     % Sort the eigenvectors columns by the eigenvalue indexes
15     evec = evec(:, i);
16
17     % PCA only works if you subtract the mean
18     result = evec' * (data - m)';
19 end
```
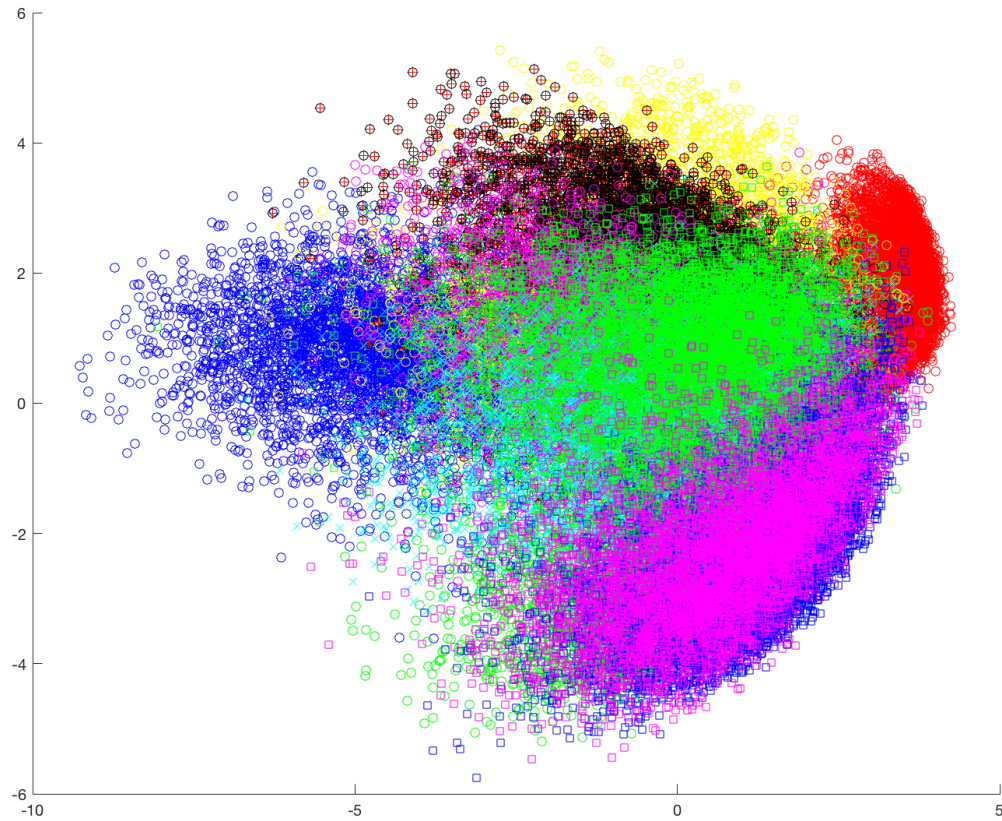
Question 5.2

a

Figure 9: PCA on MNIST dataset

b

The first principle component accounts for 9.7047% of the data, whereas the second principle component accounts for 7.0959% of the data.
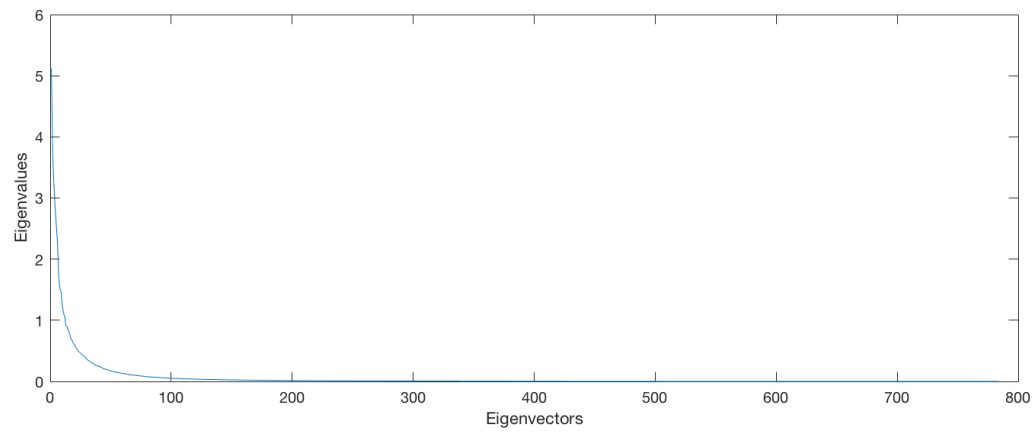
c

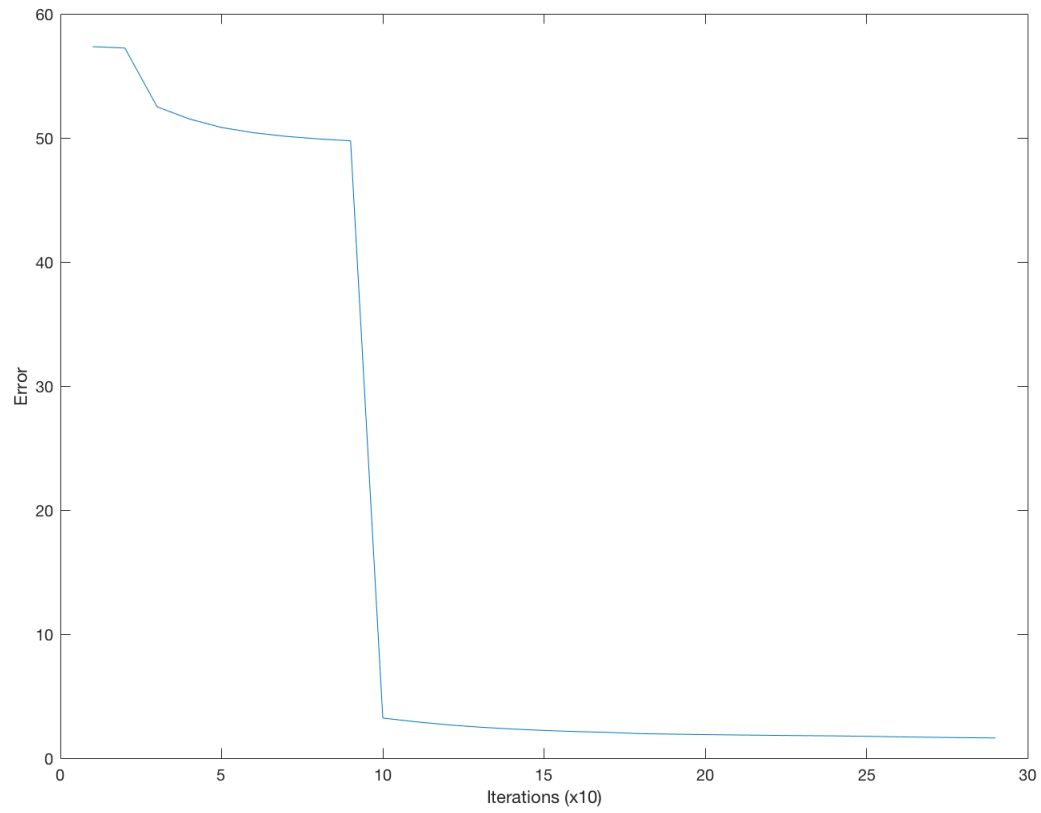Figure 10: Scree Graph

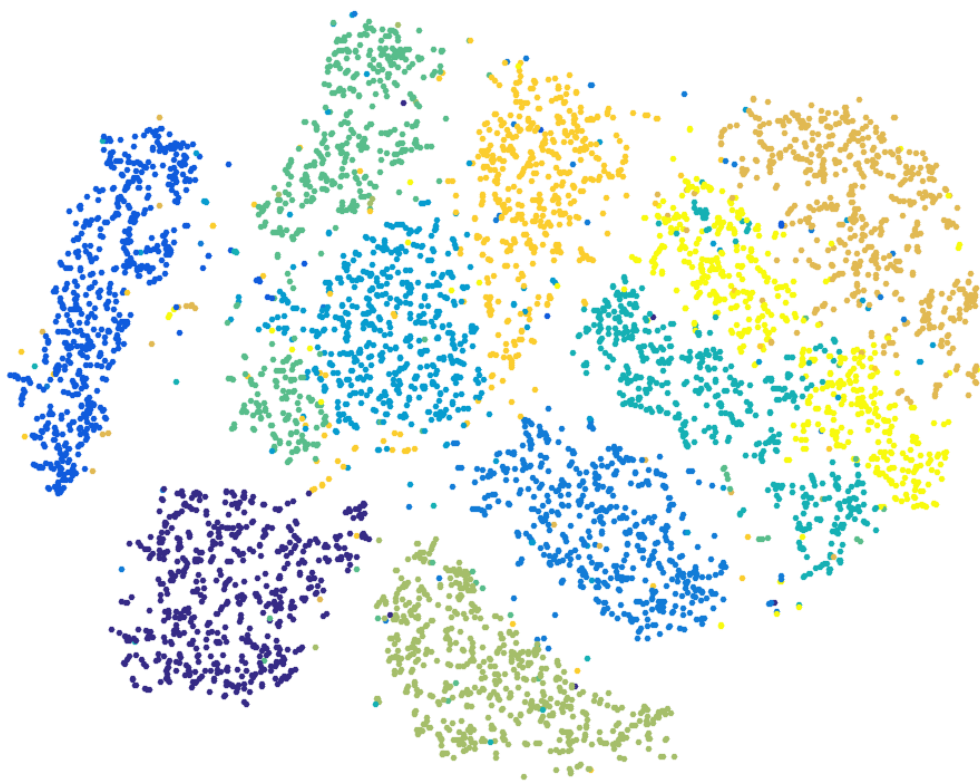# Question 5.6

Figure 11: Error vs Iteration
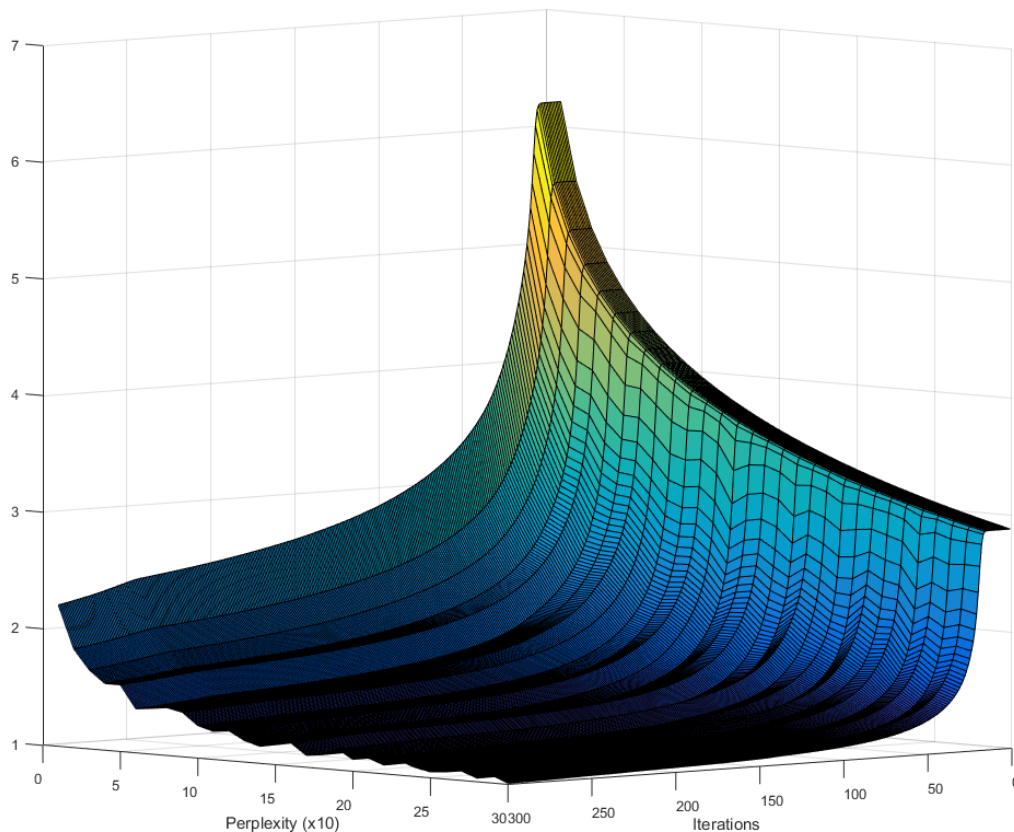
Figure 12: Iteration 300

Question 5.8

Figure 13: Perplexity and Iterations on Cost

The above graph shows the perplexity and iterations on the cost (error). From inspecting the graph it can be seen that increasing the iterations has a much larger effect than increasing the perplexity. Thus it is clear that the number iterations has a much greater influence on the data than the perplexity. Also its worth noting the graph shows a plateau when the number of iterations is small (when number of iterations is between 0 and 25). Additionally after the plateau there is a steep fall in the error as the iterations increase, with the error starting to flatten at the 100 iterations mark. The perplexity determines how to balance the attention between local and global aspects of the dataset, the original paper written on t-SNE suggests a value for the perplexity between 5 and 50[1]. From the above graph it can be seen that the cost is highest with small values for iterations and perplexity, thus those low numbers are not suitable for the heuristic.

However as the perplexity and iterations increase, the error decreases. Thus a valid heuristic would be to choose the largest number for perplexity and iterations as possible, as

this will guarantee a smaller error. There is a trade off for this method as the number of iterations will increase the computation time. Additionally perplexity should be less than the number of data points[2].
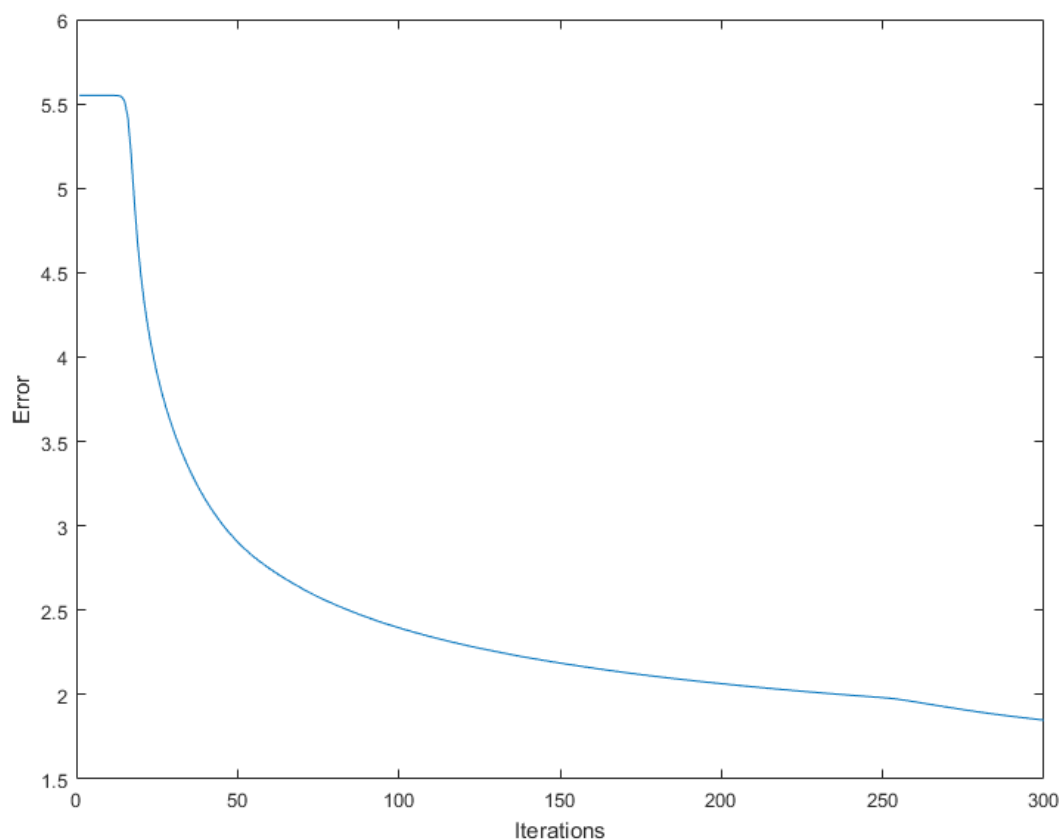

Figure 14: Error on iterations (perplexity = 2)

Figure 14 shows a slice of the 3d graph when the perplexity is 2, this illustrates the plateau from earlier and shows how the error flattens out when the number of iterations is increased.

Figure 14 differs from the figure 11 as the other chart as a large dip in the error at the 100 iteration mark. This is because of the lines that were commented out, as those lines introduce a larger momentum that decays over each iteration. Line 87-89 of `tsne_p.m` tells t-SNE to stop lying about the p-values when it reaches the 100th iteration, which explains the massive drop in error.
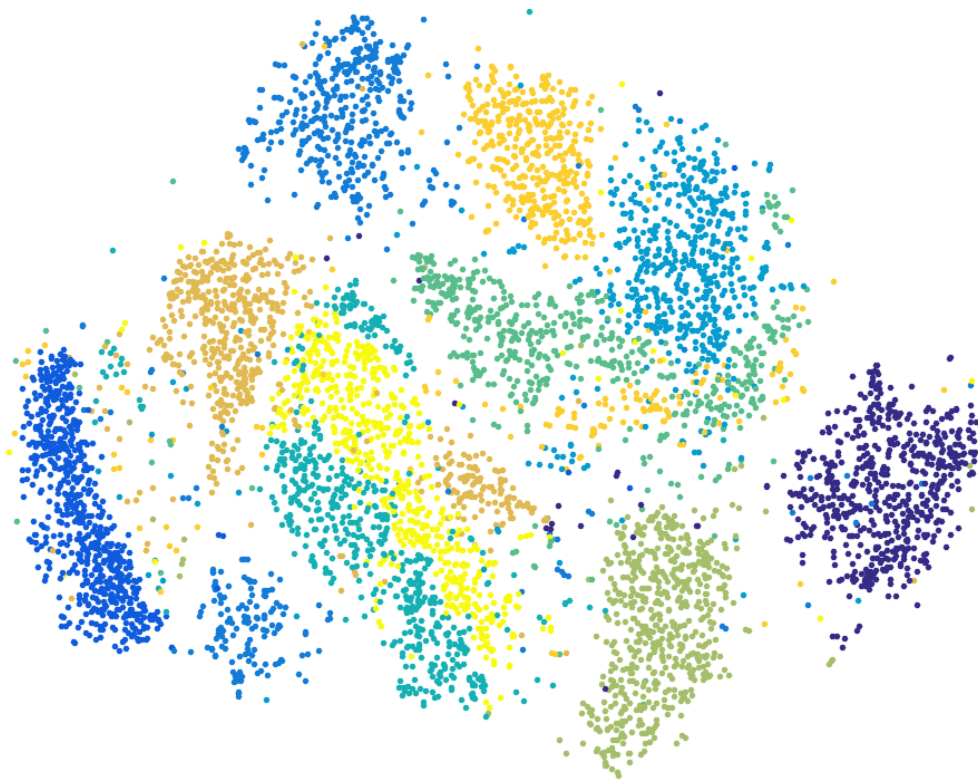
17

Figure 15: TSNE at 300 iterations

Figure 15 is similar to figure 12, however there are a few differences. The major distance is the grouping of the classes, figure 12 has much tighter grouping and better separation of classes, however as figure 12 has a lower perplexity, it could be attempting to cluster local minima.

Question 6.3

Using the code generated from the `nprtool`, the MNIST dataset can be easily loaded however there are various design choices that should be addressed. First is the separation of data into testing and training, which was done by using the given training and testing datasets from the MNIST data.

The first design choice is the network training function, Matlab supports a lot of different training algorithms, such as gradient descent backpropagation, gradient descent with momentum and scaled gradient backpropagation. The accuracy for a training algorithm

depends on a lot of things, including the dimensionality of the data, the size of the dataset and the complexity of the data. First we should look at the MNIST dataset and determine some of its traits, allowing us to make a judgement on what training algorithm to use. Below are the properties of the MNIST dataset:

- 70000 data entries

- 784 dimensions

- 10 classes

The network structure for the mnist data is `768-N-10`, where N is the number of hidden layers. The Matlab Neural Network Toolbox provides a tables of performance for different algorithms based on network structure. The closest network structure to the MNIST one is the Cholesterol dataset, which has a network structure of `21-25-3`. This dataset has a large number of inputs relative to the outputs, which makes it similar to the MNIST dataset.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
SCG	99.73	1.00	83.10	113.40	9.93
CGP	121.54	1.22	101.76	162.49	16.34
CGB	124.06	1.2	107.64	146.90	14.62
CGF	136.04	1.36	106.46	167.28	17.67
LM	261.50	2.62	103.52	398.45	102.06
OSS	268.55	2.69	197.84	372.99	56.79
BFG	550.92	5.52	471.61	676.39	46.59
RP	1519.00	15.23	581.17	2256.10	557.34
GDX	3169.50	31.78	2514.90	4168.20	610.52

Figure 16: Comparison of algorithms from Matlab documentation[3]

From the table, it can be seen that the SCG algorithm was the best, which is the Scaled Gradient Backpropagation algorithm.

I chose the scaled gradient backpropagation (`trainscg`) function as it does not require a line search on each iteration, which is described as a computationally intensive operation in the Matlab documentation. The Matlab describes the downside of adopting this function is it takes more iterations to converge than the other algorithms, but the speed increase is worth it.

Another design parameter is the number of neurons in the hidden layers. The documentation for Scikit learn states the general rule of thumb for choosing values for the hidden layers is a value between the number of neurons in the input and output layers. As we are dealing with 784 input layers and 10 output layers, any value between those would work. To choose a suitable value, the network was trained against 15, 30, 45 and 60 hidden neurons, below is a chart showing the results.
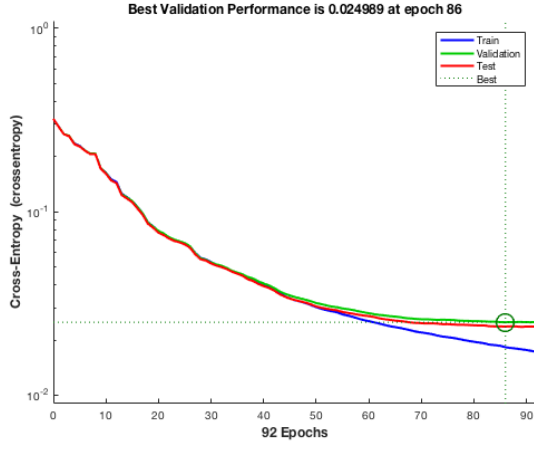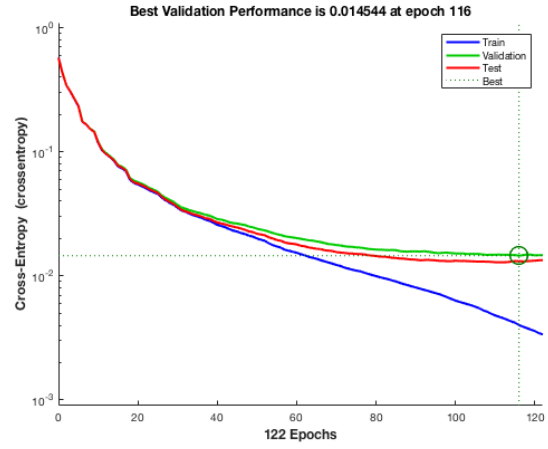
(a) 15 hidden neurons training
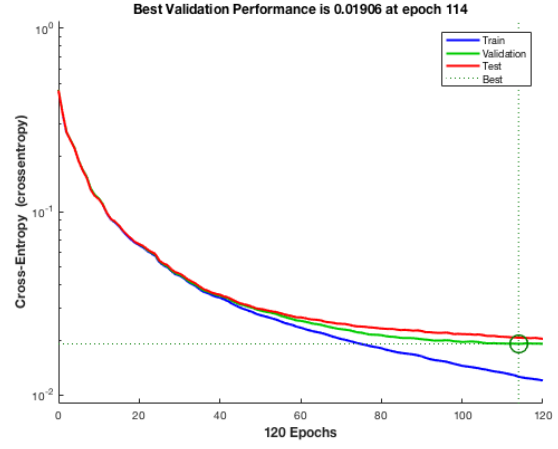
(b) 30 hidden neurons training
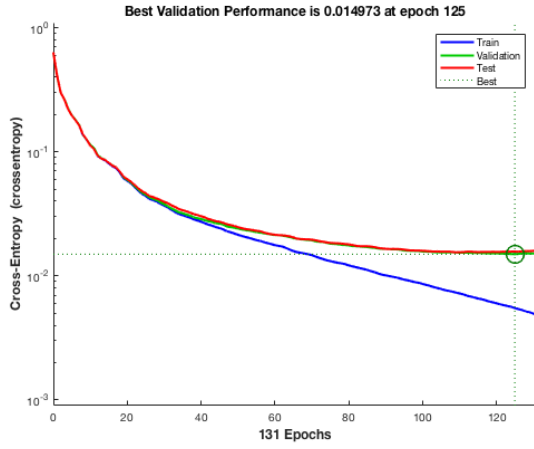
(c) 45 hidden neurons training

(d) 60 hidden neurons training

Figure 17: Training results

20

The results of the training was summarised into the following graph.
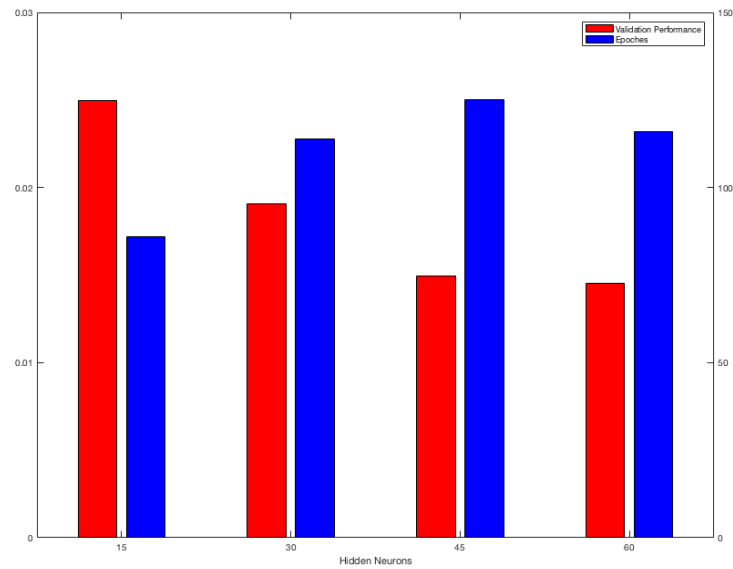

Figure 18: Number of Neurons vs Performance and Epochs

It can be seen that after 45 neurons, there are only marginal increases in performance, but the number of epochs until converging increases, thus increasing the running time of the program. As a result 60 hidden layers was selected for the dataset.
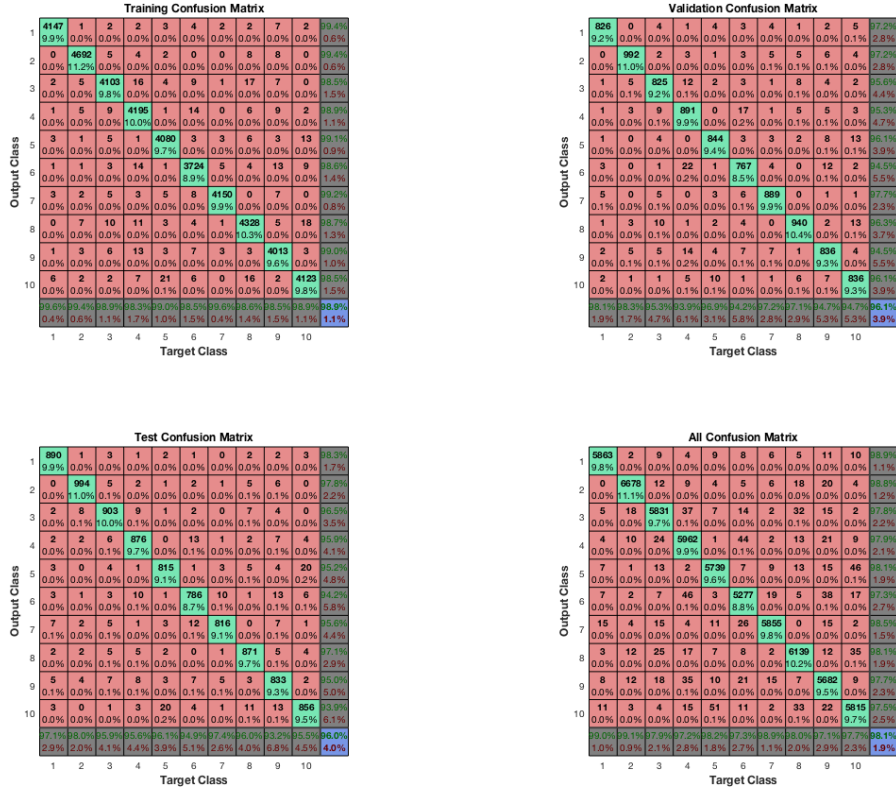
Figure 19: 60 hidden neurons confusion matrix

The above confusion matrix chart shows the error in the algorithm. The validation confusion matrix shows that the overall error is 3.9%.

This can be compared to Figure 9 in the paper "Gradient-Based Learning Applied to Document Recognition"[4], which found the error rate to be 4.7%. Its also worth noting the error on the training confusion matrix, which has by far the lowest error at 1.1% which matches the blue line from part d in figure 17.

Neural Network Code

```matlab
1  % Solve a Pattern Recognition Problem with a Neural Network
2  % Script generated by Neural Pattern Recognition app
3  % Created 28−Apr−2017 09:08:10
4  %
5
6  load('mnist_train.mat');
7  load('mnist_test.mat');
8
9  all_labels = [train_labels; test_labels];
10 all_X = [train_X; test_X];
11 labels = [];
12 for i = 1:10
13     labels = [labels, all_labels == i];
14 end
15
16 x = transpose(all_X);
17 t = transpose(labels);
18
19 % Choose a Training Function
20 % For a list of all training functions type: help nntrain
21 % 'trainlm' is usually fastest.
22 % 'trainbr' takes longer but may be better for challenging
       problems.
23 % 'trainscg' uses less memory. Suitable in low memory situations.
24
25 trainFcn = 'trainscg';  % Scaled conjugate gradient
       backpropagation.
26 %trainFcn = 'traingd';
27
28 % Create a Pattern Recognition Network
29 hiddenLayerSize = 20;
30 net = patternnet(hiddenLayerSize, trainFcn);
31
32 net.trainParam.epochs = 1000;
33 % net.trainParam.lr = 0.1;
34
35 % Setup Division of Data for Training, Validation, Testing
36 net.divideParam.trainRatio = 5/7;
37 net.divideParam.valRatio = 1/7;
38 % 1/7 of the dataset is the testing data, which is at the end
39 net.divideParam.testRatio = 1/7;
40
41 % Train the Network
```

```
42  [ net , tr ]  =  train ( net , x , t ) ;
43
44  % Test  the  Network
45  y  =  net ( x ) ;
46  e  =  gsubtract ( t , y ) ;
47  performance  =  perform ( net , t , y )
48  tind  =  vec2ind ( t ) ;
49  yind  =  vec2ind ( y ) ;
50  percentErrors  =  sum ( tind  ~=  yind ) / numel ( tind ) ;
51
52  % View  the  Network
53  view ( net )
54
55  % Plots
56  % Uncomment  these  lines  to  enable  various  plots .
57  figure ,  plotperform ( tr )
58  %figure ,  plottrainstate ( tr )
59  %figure ,  ploterrhist ( e )
60  %figure ,  plotconfusion ( t , y )
61  %figure ,  plotroc ( t , y )
62
63  percentErrors
```

References

[1] Laurens van der Maaten and Geoffrey E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[2] Martin Wattenberg, Fernanda Vigas, and Ian Johnson. How to use t-sne effectively. *Distill*, 2016.

[3] Choose a multilayer neural network training function - matlab, url: https://au.mathworks.com/help/nnet/ug/choose-a-multilayer-neural-network-training-function.html.

[4] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.