Assignment 2, COMP4702

Roy Portas, 43560846

May 4, 2017

Question 4.2

```
% Q2

a = randn(200, 2);
b = a + 4;
c = a;

c(:, 1) = 3 * c(:, 1);
c = c - 4;

d = [a; b];
e = [a; b; c];

% hold on;
% plot(a(:, 1), a(:, 2), '+');
% plot(b(:, 1), b(:, 2), 'o');
% plot(c(:, 1), c(:, 2), '*');

% Use the first dataset
data = d;


figure;
subplot(3, 2, 1);
hold on;
% ksdensity(data, 'PlotFcn', 'contour');
plot(a(:, 1), a(:, 2), '+');
plot(b(:, 1), b(:, 2), 'o');
plot(c(:, 1), c(:, 2), '*');

title('Contours Overlay');

% Calculate the grid
[f, xi] = ksdensity(data); x = linspace(min(xi(:, 1)),max(xi(:, 1)
    ));
y = linspace(min(xi(:, 2)),max(xi(:, 2)));
[xq,yq] = meshgrid(x,y);
z = griddata(xi(:, 1),xi(:, 2),f,xq,yq);

% We now have x, y, z that can be used to get the gradient at any
    point

contour(x, y, z);
```

```
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;
```

```
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.

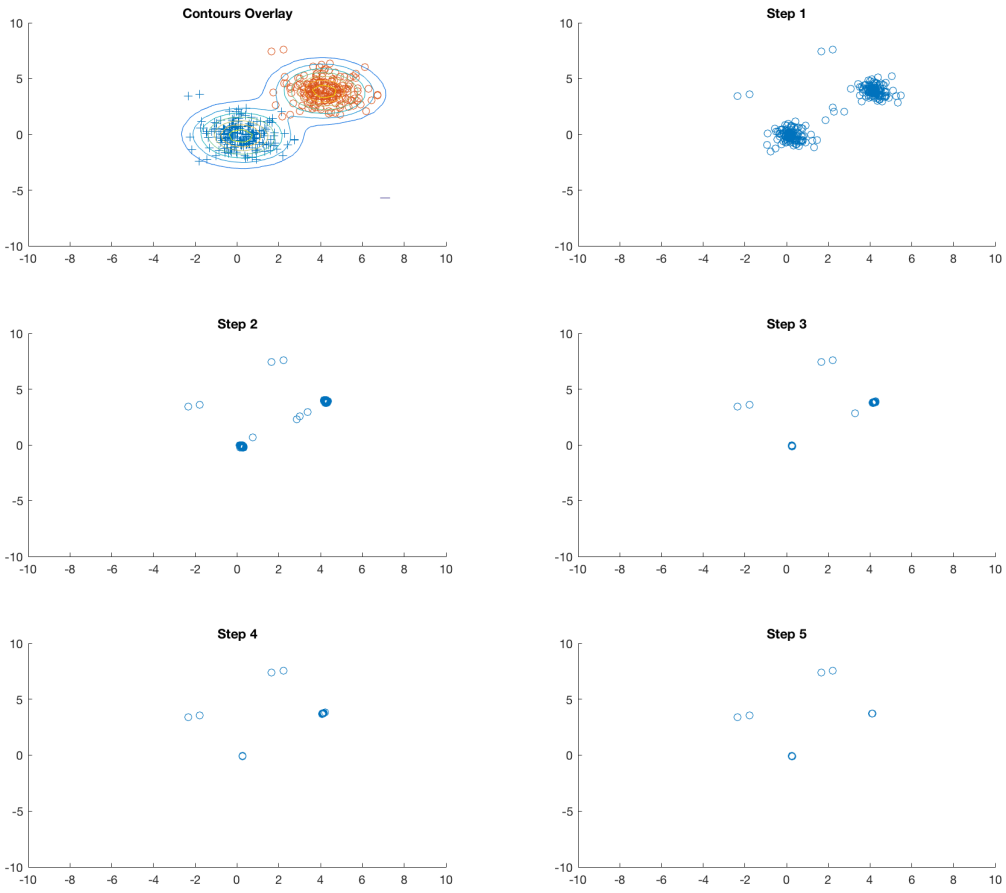


Figure 1: 2 Classes, Lambda = 1.8

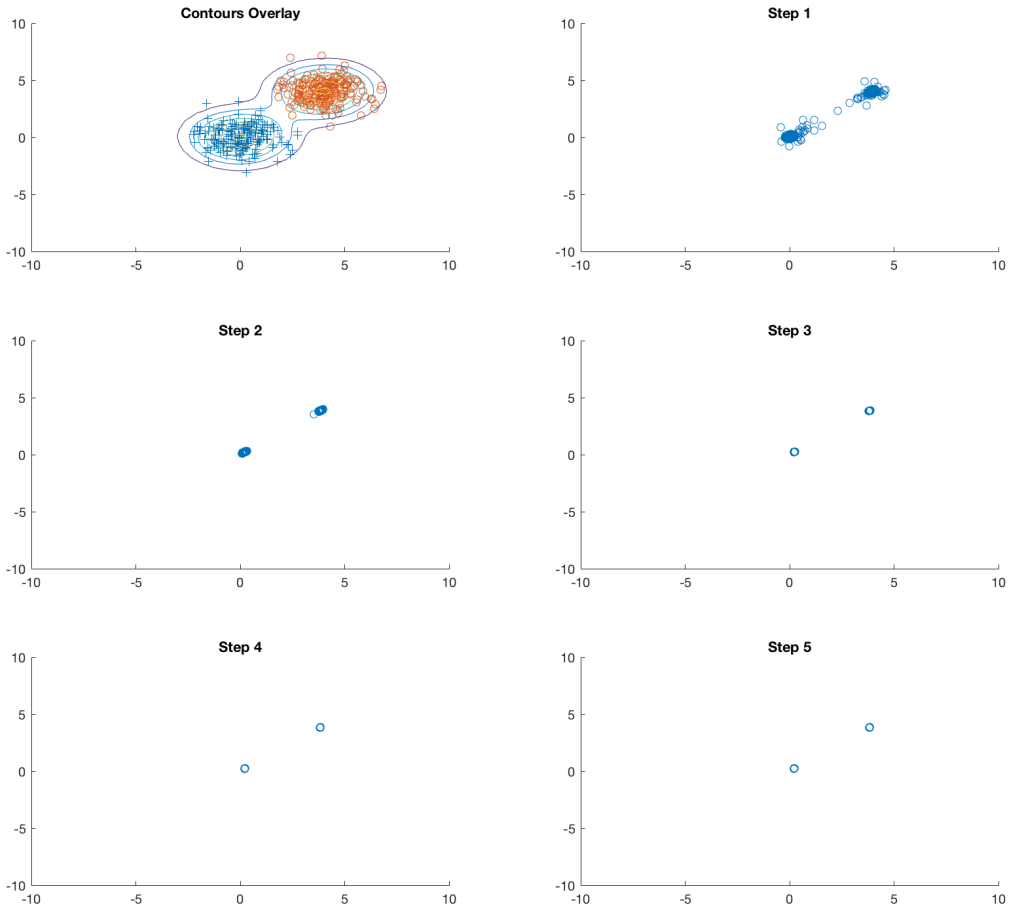


Figure 2: 2 Classes, Lambda = 3

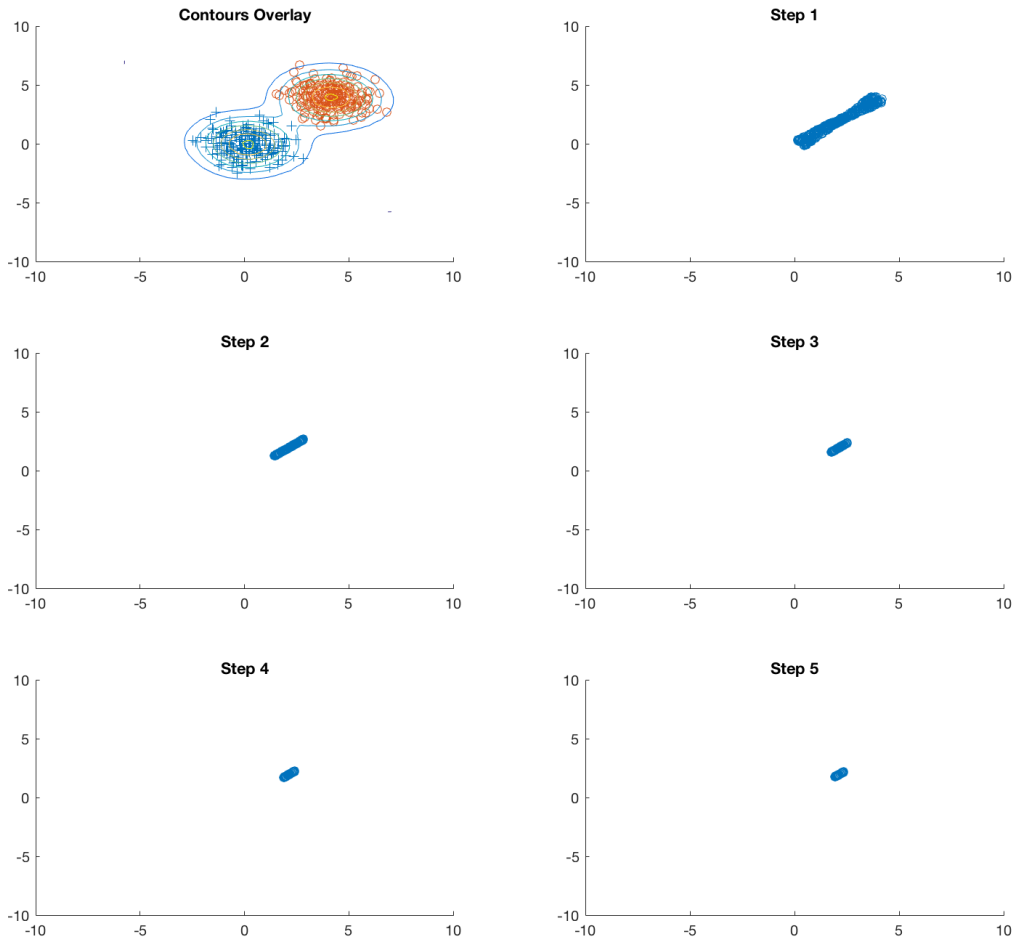


Figure 3: 2 Classes, Lambda = 5

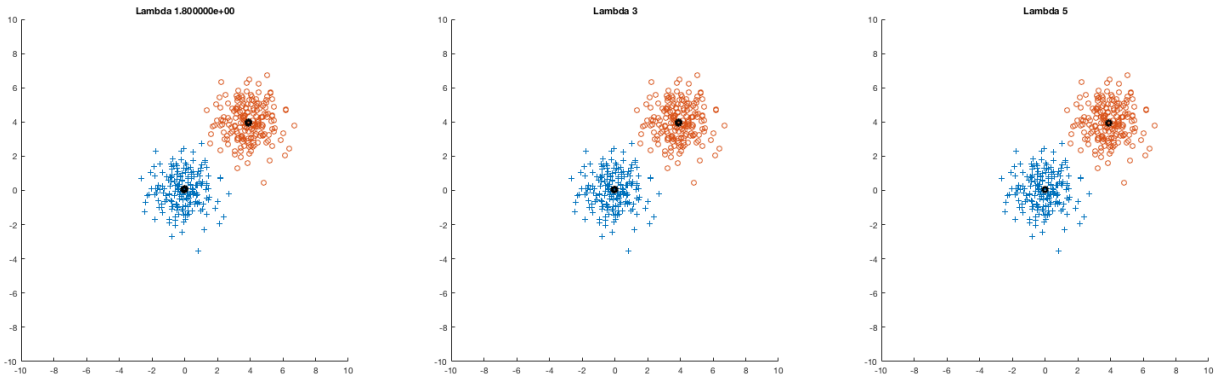


Figure 4: Cluster centres over data

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.

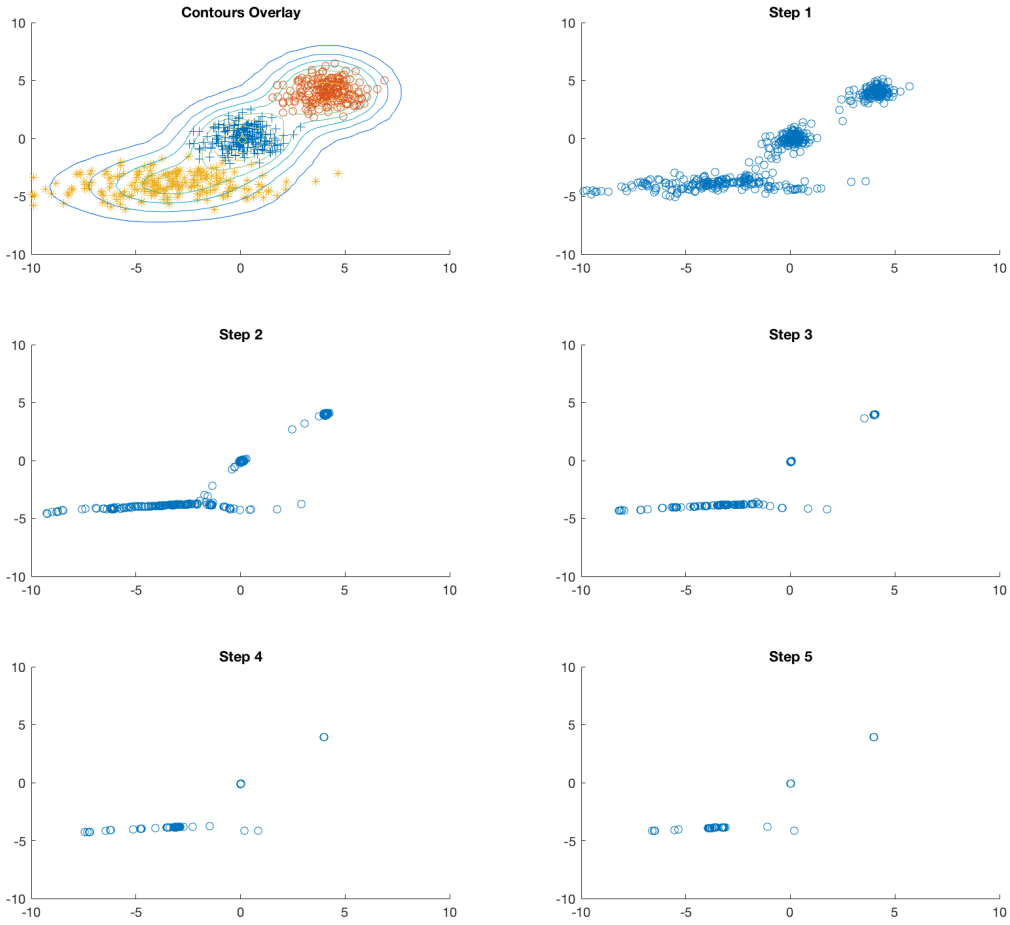


Figure 5: 3 Classes, Lambda = 1.8

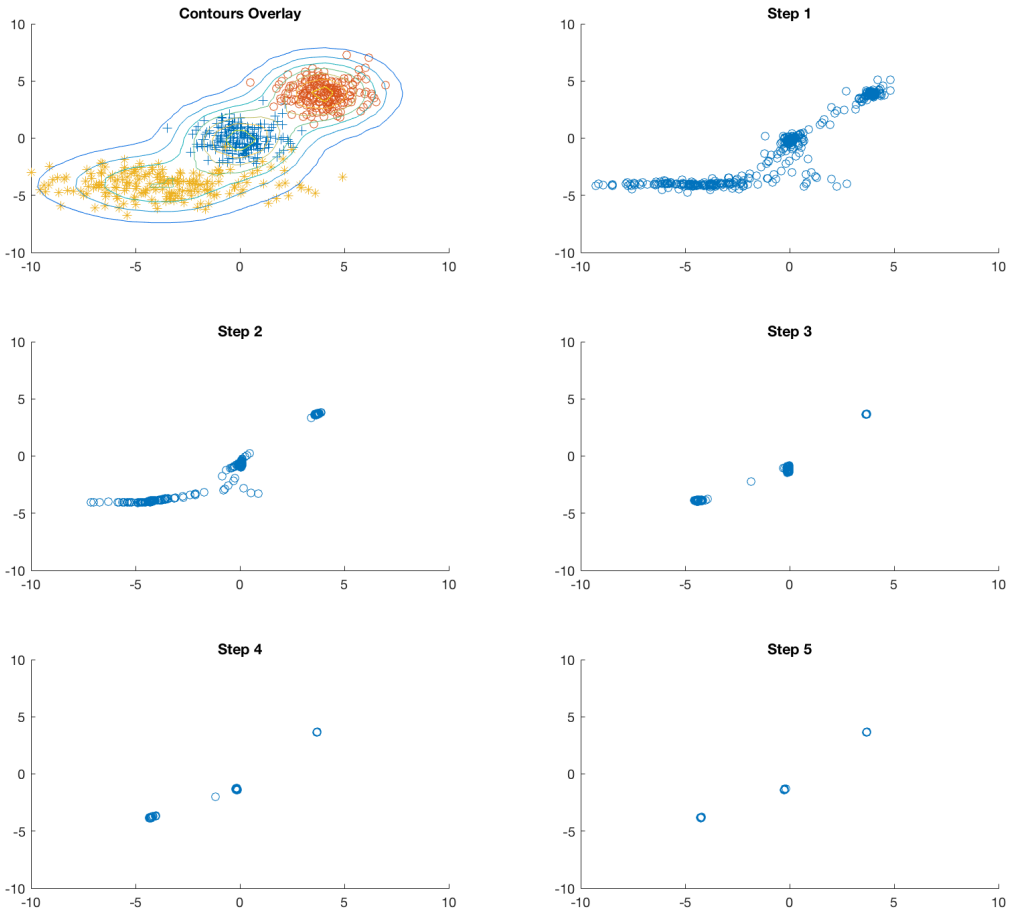


Figure 6: 3 Classes, Lambda = 3

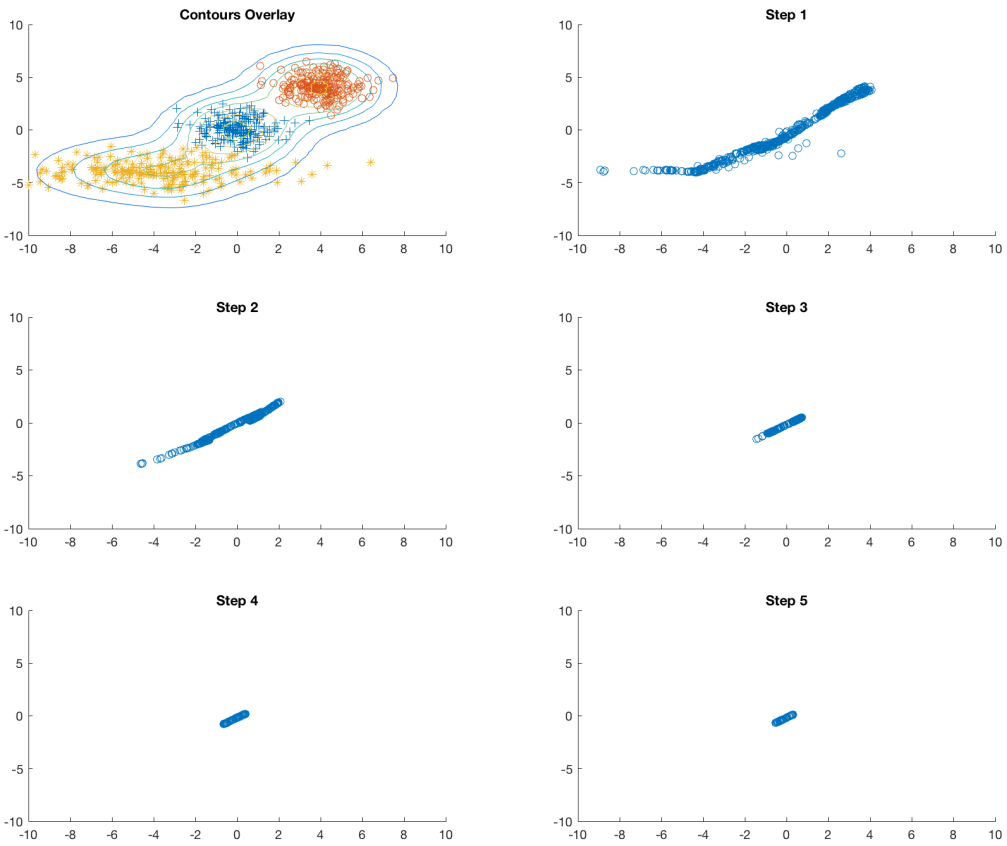


Figure 7: 3 Classes, Lambda = 5

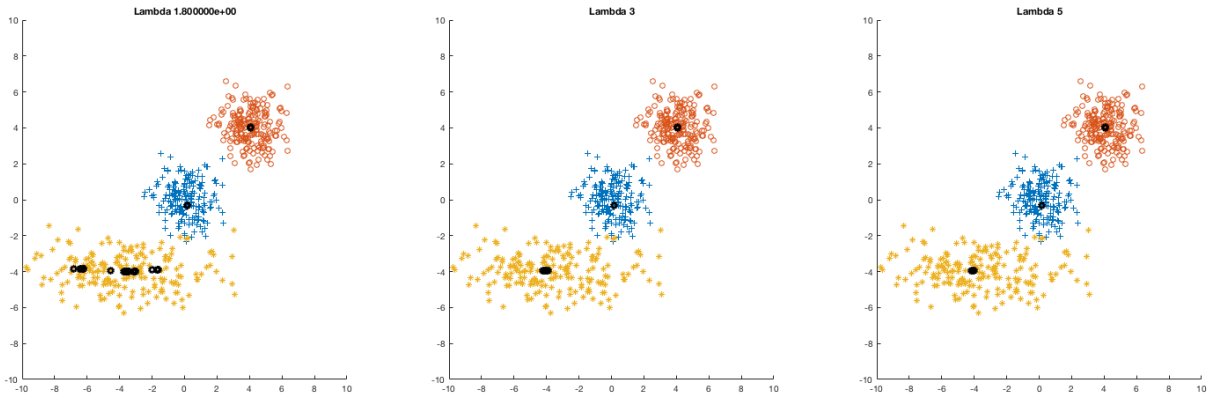


Figure 8: Cluster centres over data

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

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

```
%
% Principle Component Analysis
%
function result = pca(data)
    % Find the mean of the data
    m = mean(data);
    % Find the covariance on the normalized data
    S = cov(data - m);
    % Calculate the eigenvalues and eigenvectors of the covariance
        matrix
    [evec, eval] = eig(S);

    % Sort the eigenvalues
    [y, i] = sort(diag(eval), 'descend');
    % Sort the eigenvectors columns by the eigenvalue indexes
    evec = evec(:, i);

    % PCA only works if you subtract the mean
    result = evec' * (data - m)';
end
```

Question 5.2

a

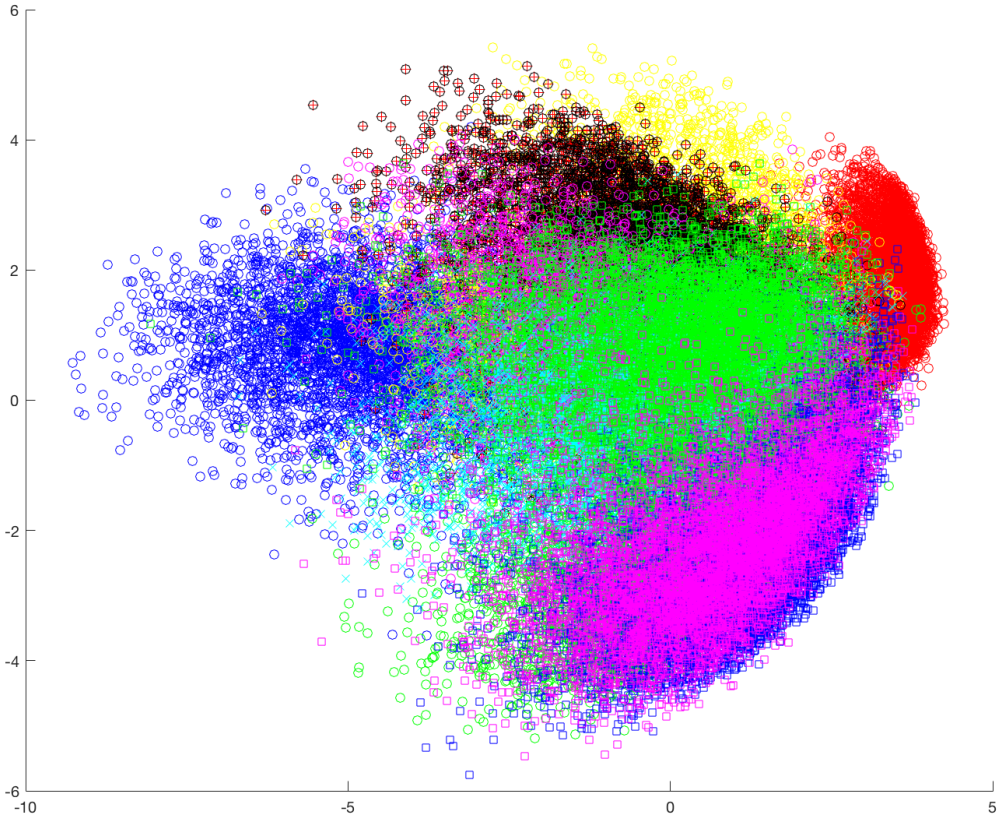


Figure 9: PCA on MNIST dataset

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

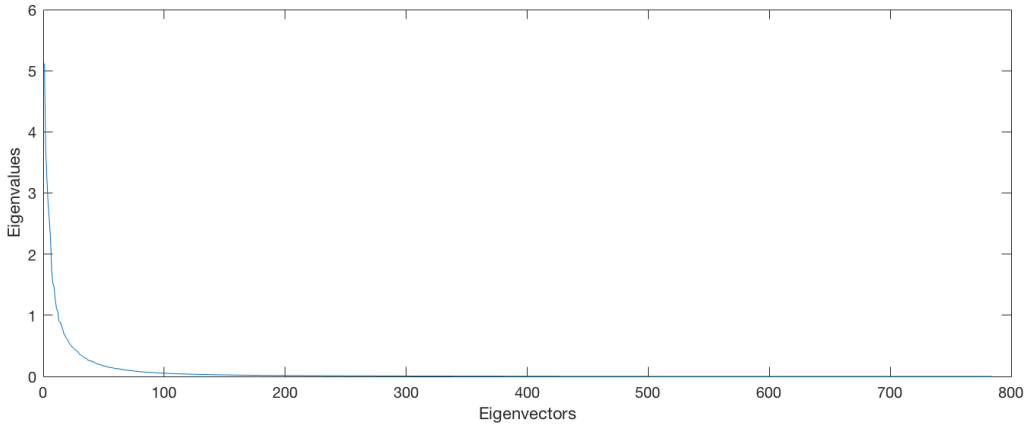


Figure 10: Scree Graph

Question 5.6

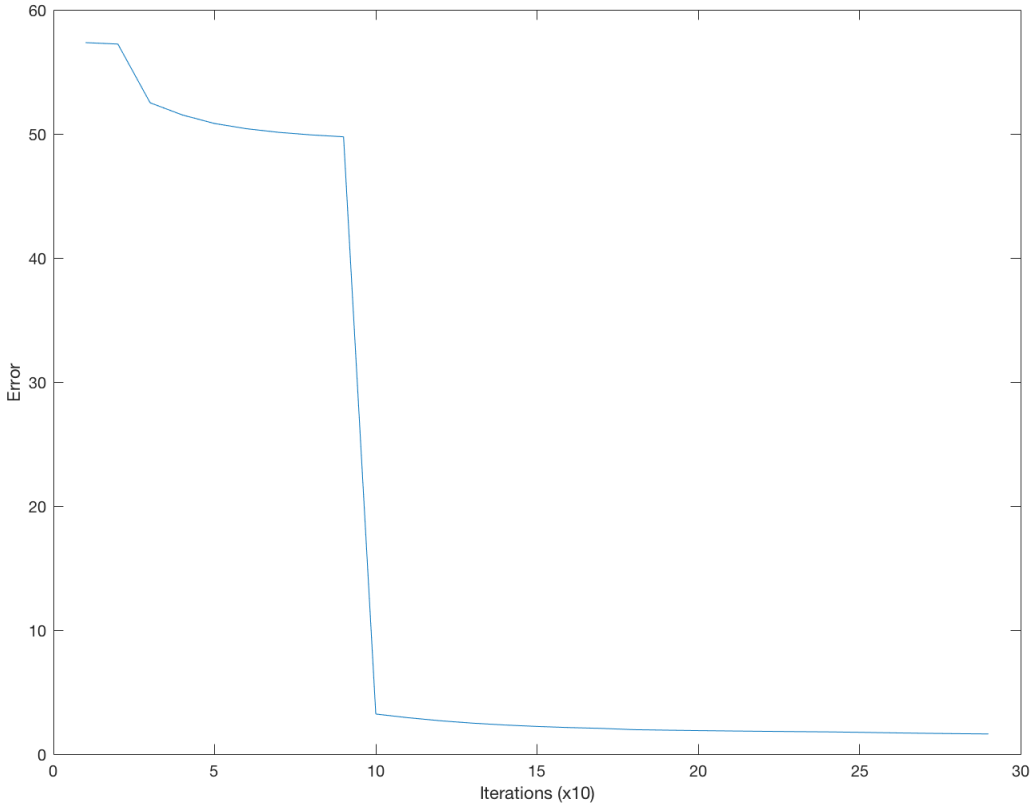


Figure 11: Error vs Iteration

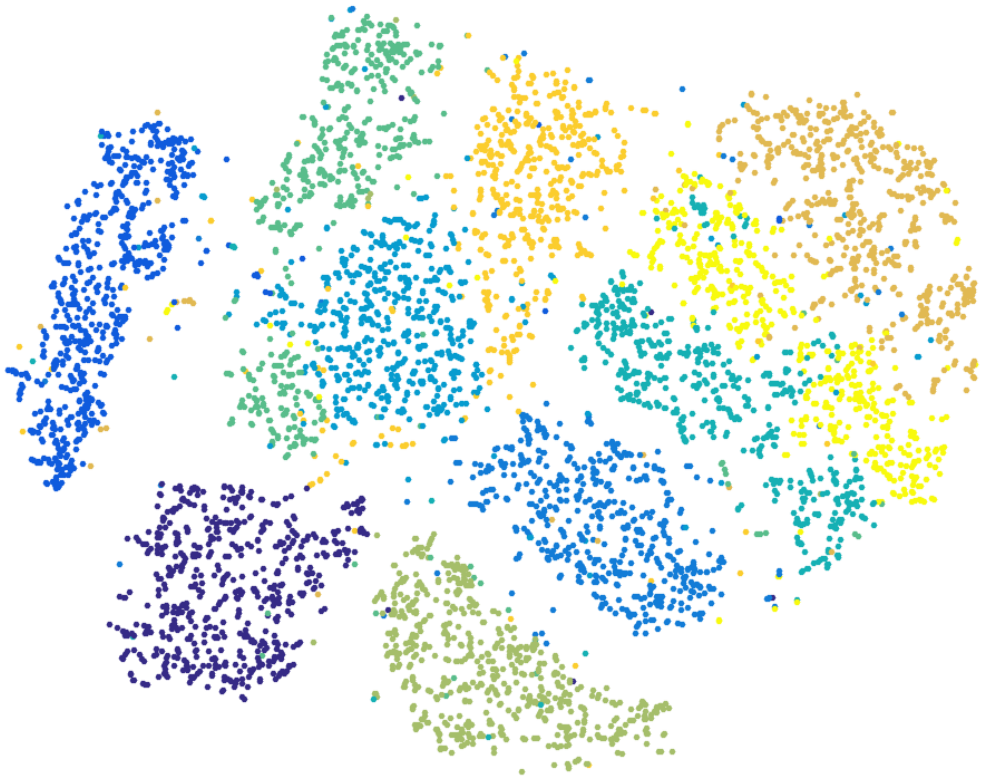


Figure 12: Iteration 300

Question 5.8

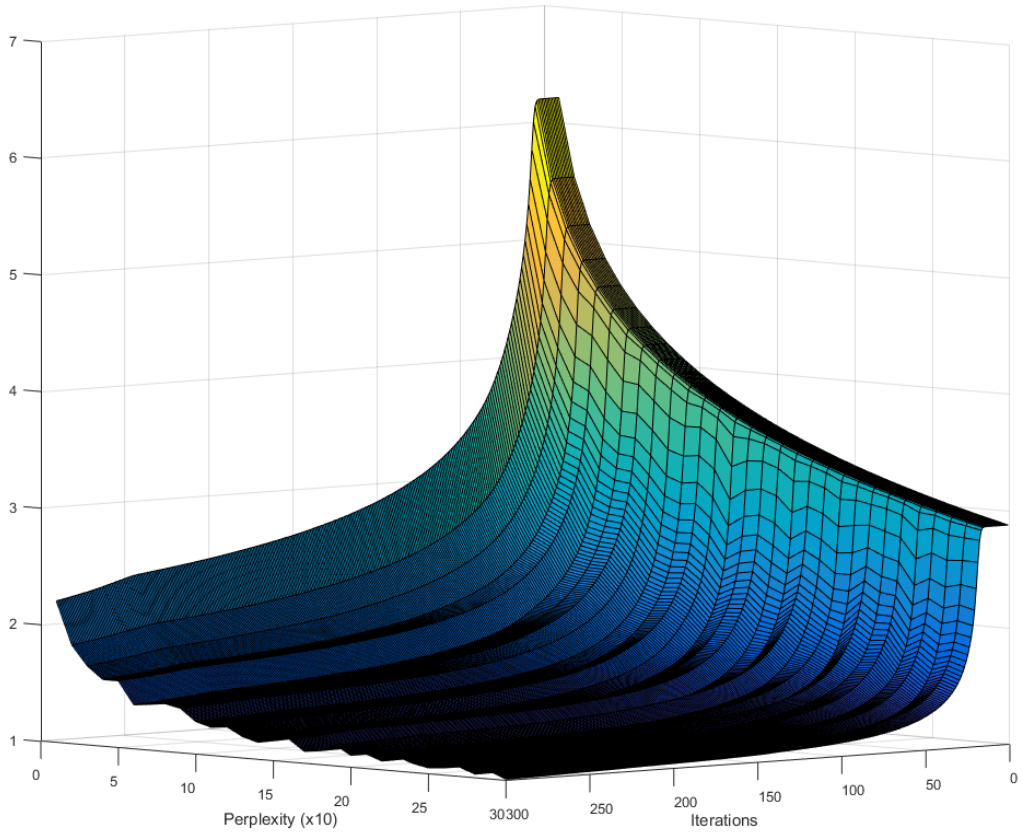


Figure 13: Perplexity and Iterations on Cost

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].

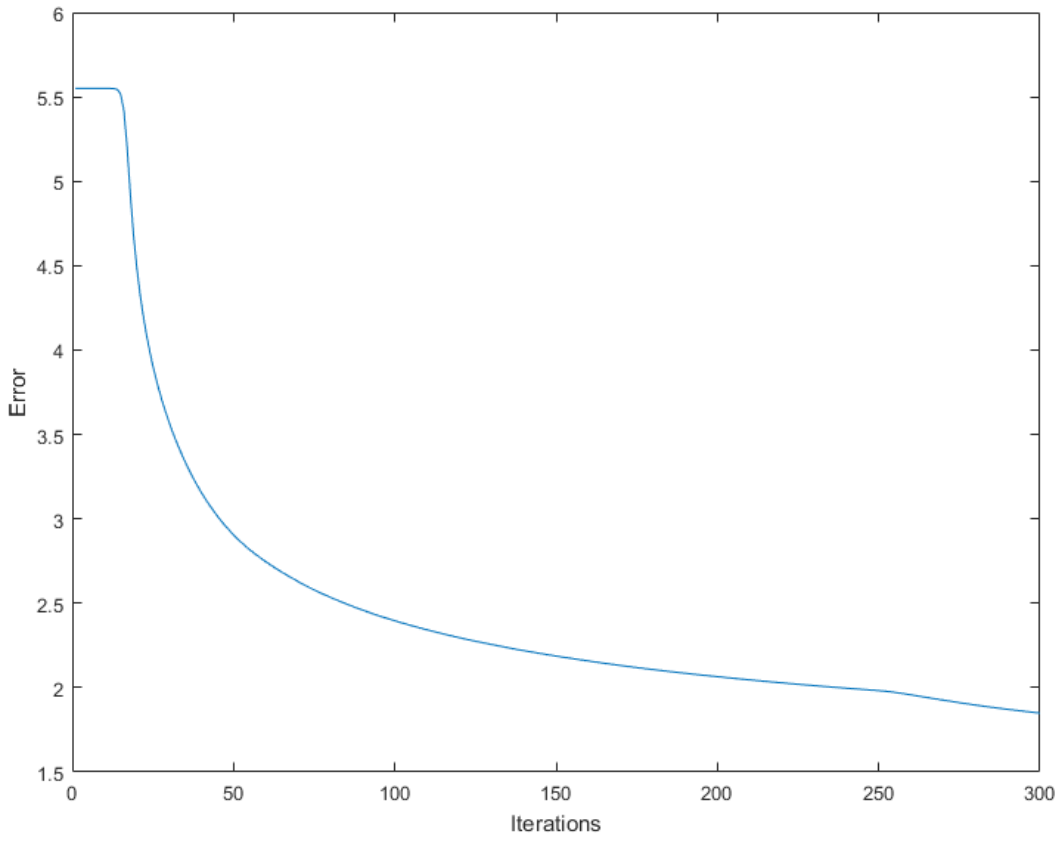


Figure 14: Error on iterations (perplexity = 2)

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.

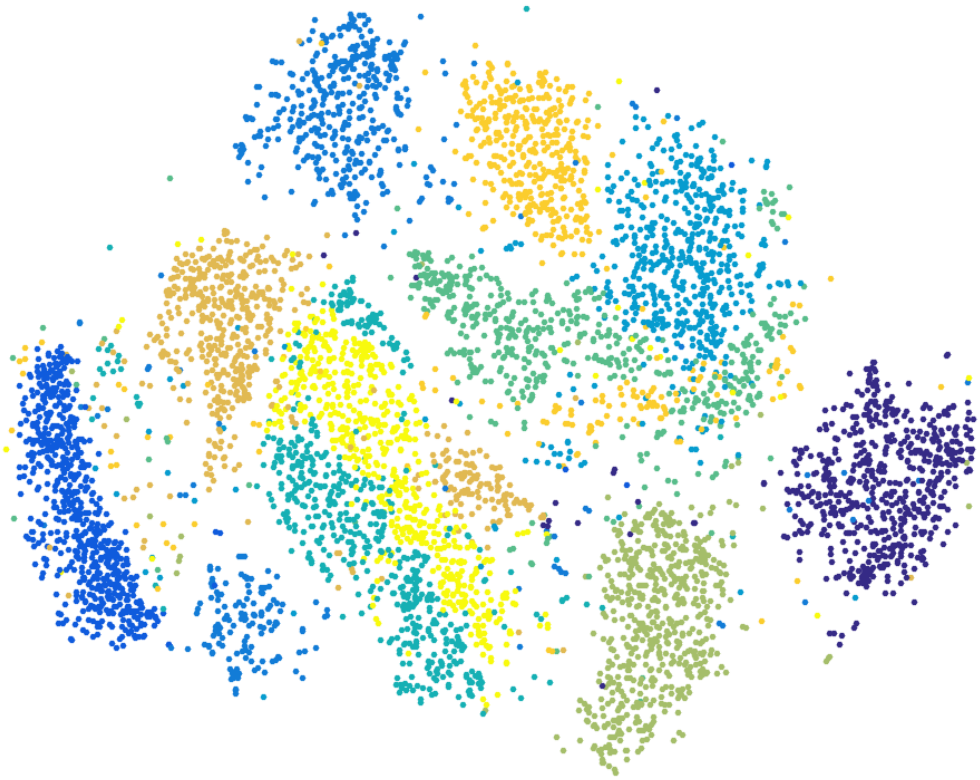


Figure 15: TSNE at 300 iterations

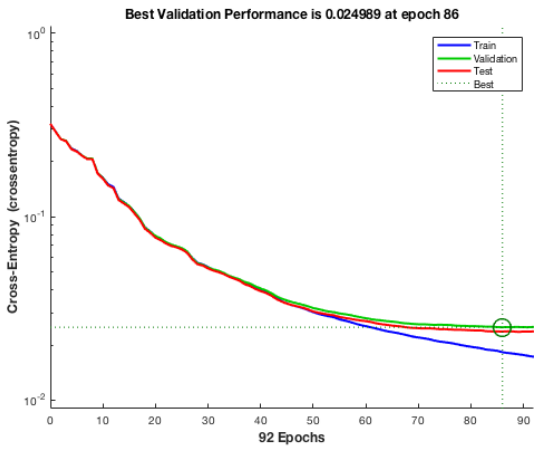
Figure 15 is similar to figure 12, however there are a few differences. The major distance is the grouping of the classes, figure 12 has much tighter grouping and better separation of classes, however as figure 12 has a lower perplexity, it could be attempting to cluster local minima.

Question 6.3

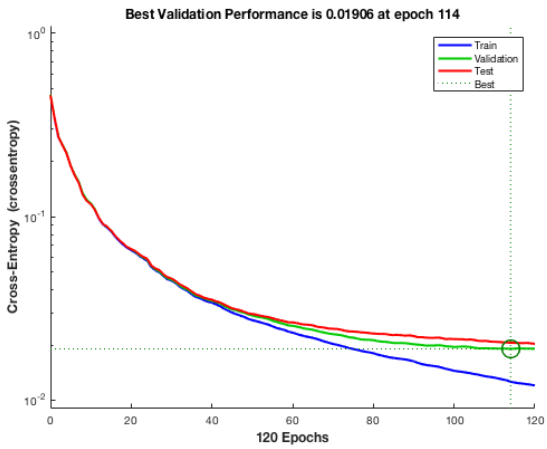
Using the code generated from the `nprtool`, the MNIST dataset can be easily loaded however there are various design choices that should be addressed.

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.

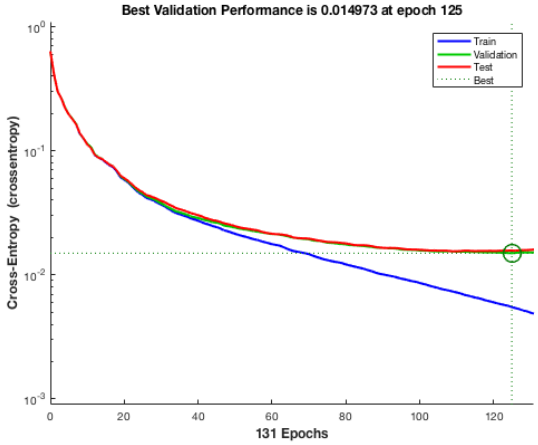
I chose the scaled gradient backpropagation (`trainscg`) function as it does not require



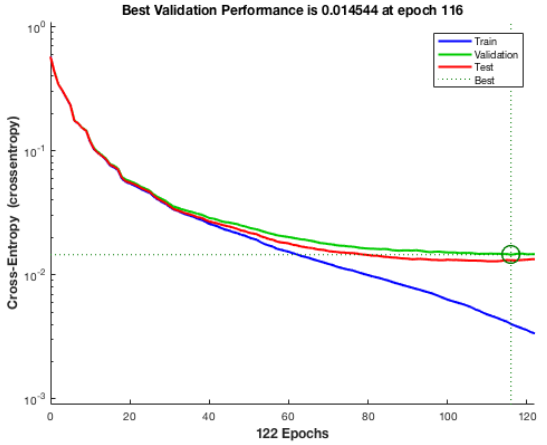
(a) 15 hidden neurons training



(b) 30 hidden neurons training



(c) 45 hidden neurons training



(d) 60 hidden neurons training

Figure 16: Training results

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.

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

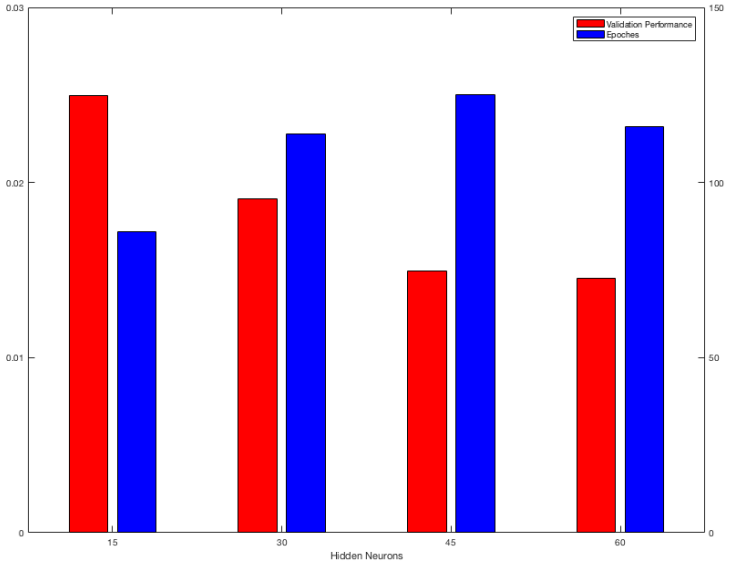


Figure 17: Number of Neurons vs Performance and Epochs

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.

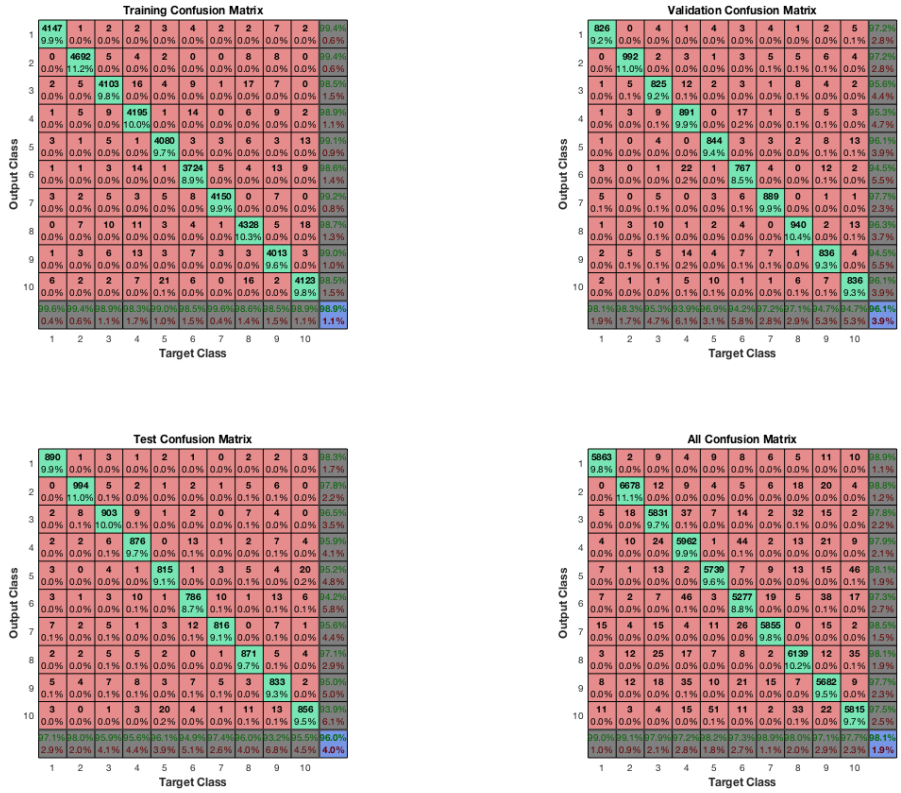


Figure 18: 60 hidden neurons confusion matrix

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"[3], 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 16.

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] 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.