Question 1:

1.1: The 8 puzzle can be modelled as a graph in the following way. Each configuration of the puzzle is represented by a node, and two nodes are connected if they are one move away from each other. For example:

	1	2
3	4	5
6	7	8

and

1		2
3	4	5
6	7	8

are connected, because one can be obtained from the other by switching the blank tile with the 1 tile. The distance from one node to another can be represented as the number of moves it takes to get from one configuration to another.

In order to solve the puzzle, we need to find a path through the graph from the start node, to the goal node, which we will take as:

	1	2
3	4	5
6	7	8

To represent this in code, we will represent each configuration of a 3x3 board as a 9-tuple, with 0 representing the blank tile. For example, the above goal state will be represented as (0,1,2,3,4,5,6,7,8).

We can 'expand' a node by finding all the nodes that it is connected to, i.e. by making all of the moves we can from that node. For example, if we were to expand:

1		2
3	4	5
6	7	8

We would get:

	1	2
3	4	5
6	7	8

1	4	2
3		5
6	7	8

1	2	
3	4	5
6	7	8

As its children.

We can also define distance heuristics between different configurations, which are explained in a later section.

The combination of all of these attributes mean that we can model solving the 8-puzzle as solving a search problem, specifically with the A* algorithm.

1.2:

1) In the A* algorithm, we have a start node and a goal node. To each node we assign a g (distance from the start node), h (heuristic distance to the end node) and f (=g+h) value.

We proceed by the following iterative algorithm:

- Find the node that we can 'see' with the smallest f value.
- If this node is the goal node, break.
- Expand that node.
- For each child of the node:
    - Set new cost = the cost of the child node+the weight of the arc between the parent and the child node.
    - If the node is in the open list (we can 'see' it)
        - If the new cost is less than the existing cost of the child node, replace the old cost with the new cost.
    - Otherwise add it to the open list
    - Set the best parent of the child node to the parent.
- If the open list is empty, exit without error (we have expanded every node and haven't found a solution)

2) a) Define the heuristic distance as the Manhattan distance to the solution. This means that we find the distance that each tile will have to 'travel', without diagonals, to its place in the goal state. For example, if we consider only the 1 tile in this configuration:

		1

We see that the Manhattan distance is 2, because the path that the tile has to take

		1

V

	1	

V

	1	

Has a distance of 2.

If we sum the Manhattan distance of each tile, then we find the total Manhattan distance. This is admissible, as the Manhattan distance is always less than or equal to the distance that each tile will travel in the solution. This is because the Manhattan calculation assumes that the tile encounters no other tiles that need to move along its path.

b) Define the heuristic distance to be the Euclidean distance to the solution. This means we take the Euclidean (straight-line) distance between each starting tile and For example,

	1	2
4	3	5
6	7	8

Has a distance of 2, because the 3 tile is 1 unit away from the solution, as is the 4. Similarly,

	3	2
1	4	5
6	7	8

Has a distance of 2sqrt(2), as both the 1 and the 3 are sqrt(2) units away from where they should be. This is admissible because if a solution is out of place, the number of times that a square needs to move to be in the correct place will always be greater than or equal to the straight line distance to the solution tile.

I have chosen these heuristic functions because they are both simple to implement, and are both calculated in O(n), where n is the number of tiles. Because they are similar in this way, the comparison between their efficiency will be completely down to how good they are as a heuristic function.

3) I have used the same start and goal configuration as given in the specification. This code can be found in astar.py.

4) Using the start and end condition stated in the specification, I have used the python time module to time the execution of both algorithms.

Using the Manhattan heuristic, the algorithm executed in 3.17 seconds, whereas the Euclidean heuristic completed in 12.95 seconds. This should be expected, as we know that the best heuristic should be as close to the actual distance from the goal configuration as possible while still underestimating the function.

Since both heuristics are admissible, they certainly underestimate the actual distance. Therefore, the greater of the two will be the better heuristic. We now show that the Manhattan distance is in fact greater than the Euclidean distance.

Manhattan distance = $\sum_{i=0}^{9} M$ where $\left(|g_{ix} - a_{ix}| + |g_{iy} - a_{iy}|\right)$ and g is the goal state, and a is the current state.

Euclidean distance = $\sum_{i=0}^{9} E$ where $E = \sqrt{(g_{ix} - a_{ix})^2 + \left(g_{iy} - a_{iy}\right)^2}$

If we consider the parts of each summation, and square both sides:

$$E^2 = (g_{ix} - a_{ix})^2 + \left(g_{iy} - a_{iy}\right)^2$$

$$M^2 = (g_{ix} - a_{ix})^2 + 2|g_{ix} - a_{ix}||g_{iy} - a_{iy}| + (g_{iy} - a_{iy})^2$$

And so clearly $M^2 \geq E^2$, and therefore $M \geq E$ so the Manhattan distance is always greater than or equal to the Euclidean distance.

To show this, I looked at the length of the open list for each heuristic. I ran the algorithm on a number of different starting configurations, and recorded the maximum length of the open list for each one. The average maximum length of the open list for the Manhattan heuristic was 1798, and the average maximum length for the Euclidean heuristic was 4027. This means that, on average, the Euclidean heuristic required about 2.25x the number of nodes to be expanded as the Manhattan heuristic. This is due to the difference in branching factor required by each heuristic.

1.3:

The code for this question can be found in astar_general.py

We cannot take any pair of permutations.

In order to solve an 8-puzzle, we can only swap a tile with the blank tile. If we swap two tiles, this is called an inversion. We count the number of inversions by swapping tiles that are in descending order, thereby aiming to end up with this configuration:

	1	2
4	3	5
6	7	8

We call the number of inversions the parity of the configuration. By moving a tile horizontally, we do not change the number of inversions, and by moving a tile vertically, we can only increase or decrease the number of inversions by 2. Therefore we can never change the parity of the number of inversions.

We can now show that a configuration is solvable if and only if the parity of both the start and goal configuration is the same.

Question 2:

2.1: K-means clustering is an iterative algorithm with two steps. Before the iteration, we start by randomly assigning each cluster a centre, which is a vector in R^n, where n is the number of dimensions we are working in. E.g. for an 8x8 image, we are working in R^{64}.

We then repeat the following to steps:

1) We work through each point in our dataset, and assign it to the closest cluster. This is done by taking the norm of the difference of the vectors.
2) Then, for each cluster centre, we re-assign the value of the cluster to be the average value of each point that has been assigned to it.

By repeating this, the centre of each cluster becomes more accurate with each iteration. After many iterations, we can assign new points to a cluster by finding the closest centre.

In handwritten digit recognition, the vectors we are using are vectors of the brightness of the pixels in each image. In this specific example, we are working with n=64, as there are 64 pixels in each image. We are therefore trying to find a vector in R^{64} that represents each digit, and to classify a new image by finding the centroid that it is closest to.

2.2

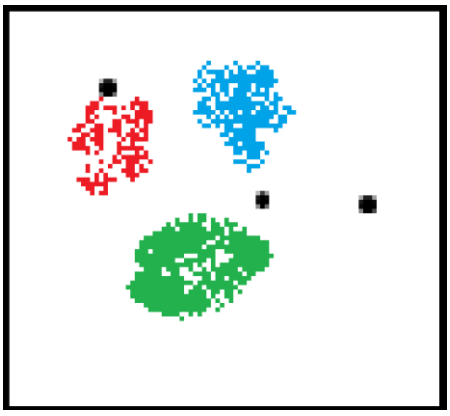
A) The code for this question can be found in k-means.py.

B) I have implemented my own version of k-means.

C) I chose to investigate the effect of the method of choosing initial centroids affects the accuracy of the classification. To start, I used random centroid selection, and ran 50 iterations of the algorithm. I then tested the accuracy of the algorithm by attempting to classify each digit in the dataset, and checking which were correct by using the sklearn datasets' labels feature. Using random clusters, the algorithm achieved an accuracy of around 55%, which is far from ideal. There are a number of reasons for this.

Firstly, when the clusters are generated randomly, there is a good chance that they are clumped together, and do not achieve a good representation of the whole dataset. This means that, if there are too few iterations, then the centroids do not end up near the right place.

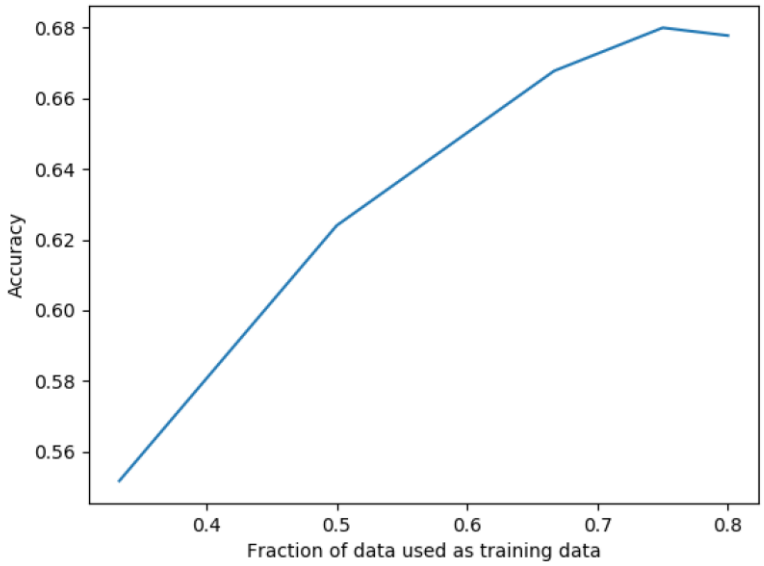
There is also a chance that one or more centroids do not get any points assigned to them. For example, if our dataset and randomised centroids look similar to this:



Then we can see that the furthest right centroid is not the closest to any of the points, and this will hugely impact our accuracy. One way of fixing this is to reselect a random cluster every time that a cluster doesn't get any points assigned to it. However, this still runs the risk of not being close to any clusters, or being too close to another centroid.

Due to the fact that we know what the clusters in this data should be at the beginning of our analysis (each cluster should be clusters of each digit, e.g. all of the 1's, 2's, 3's, etc.), we know where the centroids should roughly end up. This means that we can start our centroids off in a better location. For example, we can select one of each digit to be our starting centroid. By implementing this, the accuracy is increased to around 69%.

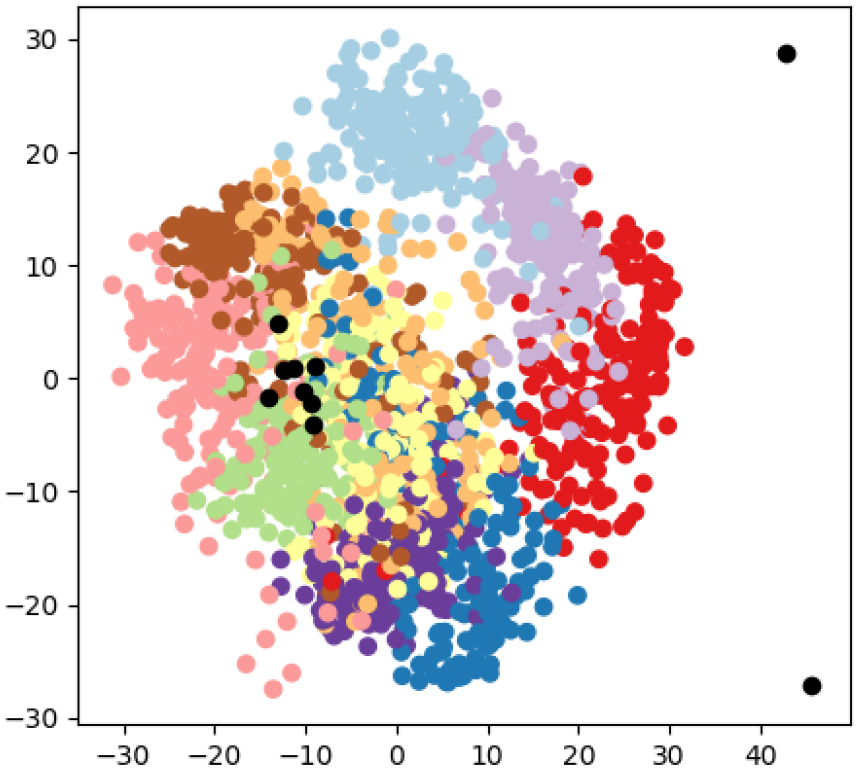
I have also investigated the effect of changing the proportion of data used as training data. I plotted this using matplotlib.pyplot, and found these results:

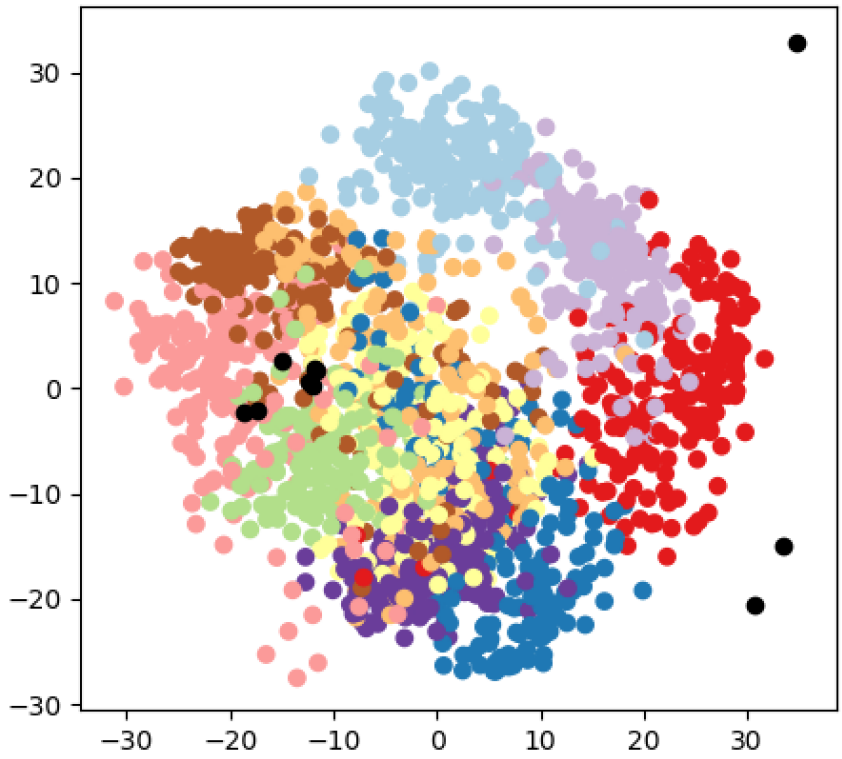


As we can see, the accuracy of the data increases with fraction of data used as training data, with a maximum accuracy of around 69%. This is as expected, as the more data we use, the more refined the clusters become. We are not expecting much higher accuracy, as there are a number of factors which will reduce it. Firstly, the data is only an 8x8 resolution, so there is less room for distinction between digits than in higher resolution images. For example, in low resolution, it could be difficult to tell the difference between a 4 and a 9, or in some cases a 1 and a 7. Digits can also be drawn in different ways, for example a 7 can be drawn as '7' or '~~7~~'. This can make it very difficult to cluster the data.

I also plotted the clusters, using the sklearn.manifold PCA dimension reduction method. This is used to represent higher dimensions, for example the 64 that we need to represent in the cluster example in a 2D plot, so we can see how well the clusters have been formed. In these plots, the black points represent the centroids, and the coloured sections represent the clusters of data. As we can see, the data overlaps a lot, but the clusters are fairly circular, and of similar size.

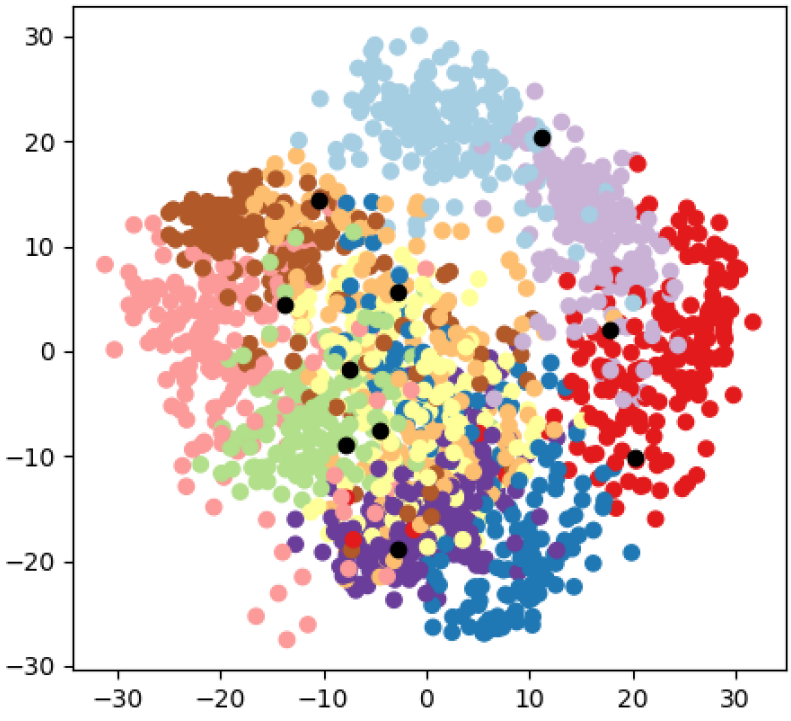
When using random clusters, we can see that the above argument is demonstrated.





In each of these images, some of the clusters are very bunched together, and some are very far away from everything, so there is very little chance that they will predict anything accurately.

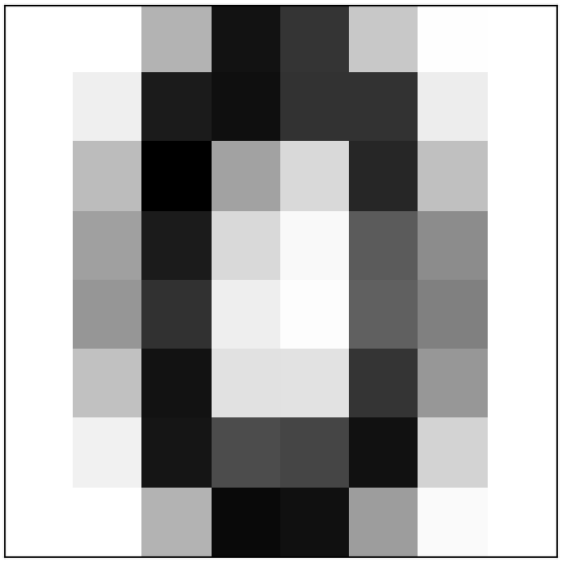
However, when we plot the same graph, with the new targeted clusters, we find that the positioning of the clusters is much better:

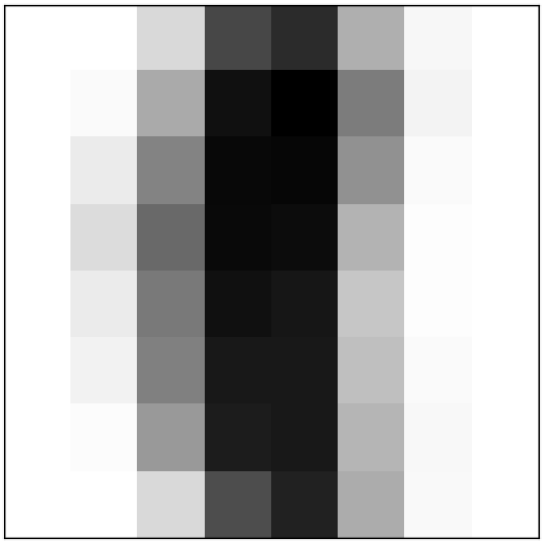


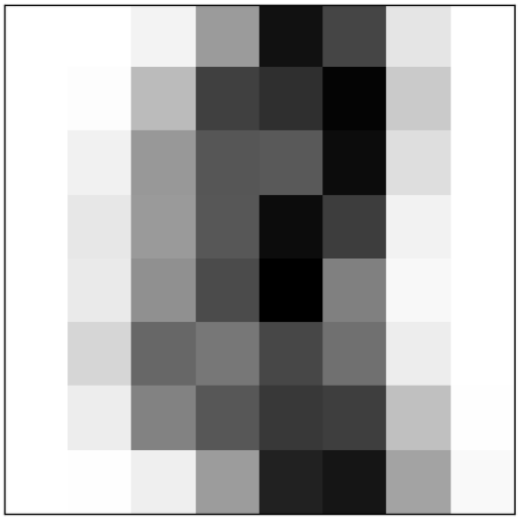
Here, the clusters are clearly more spread out, and more central to a cluster.

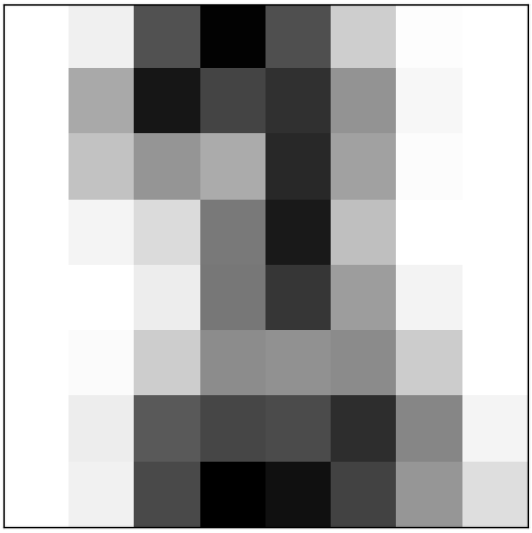
Clearly, the data points may be slightly off, because we are reducing the data from R^{64} to R^2, which is a big difference, and we can't represent as many directions for the clusters to be separate.

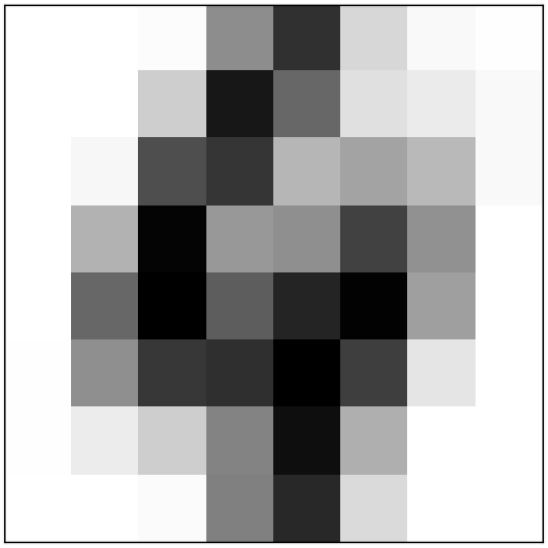
I have also displayed the centroids as images:

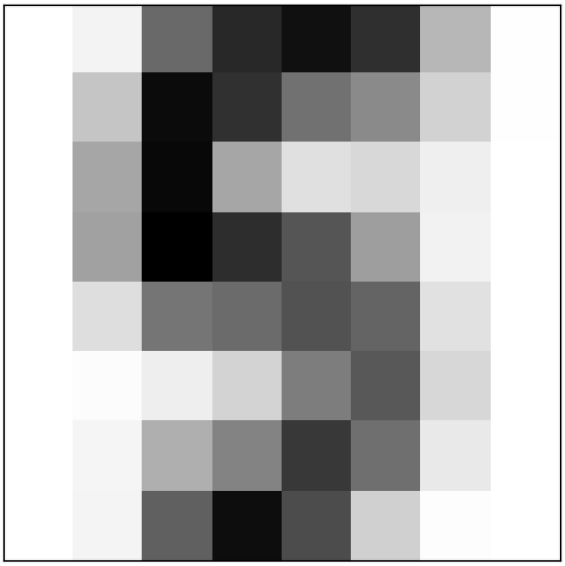


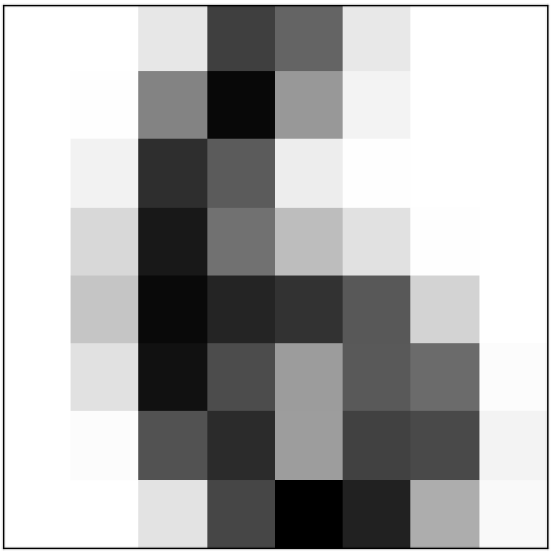


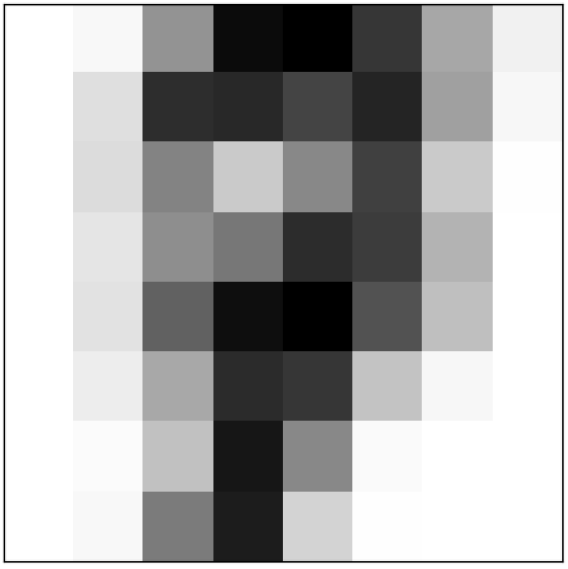


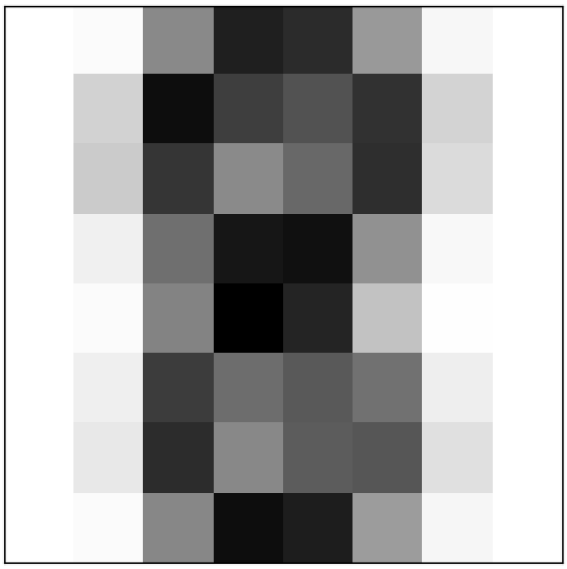


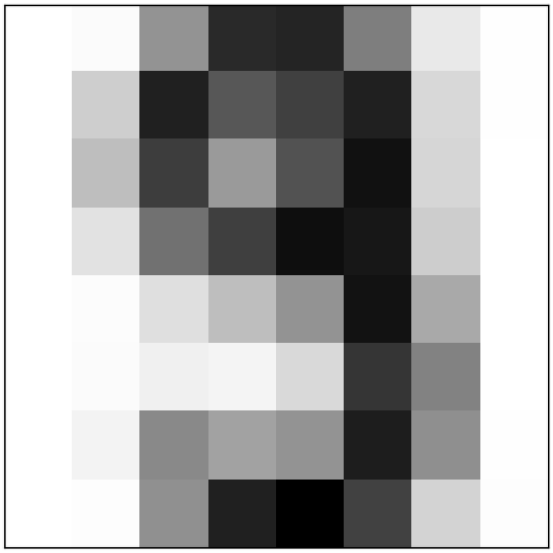








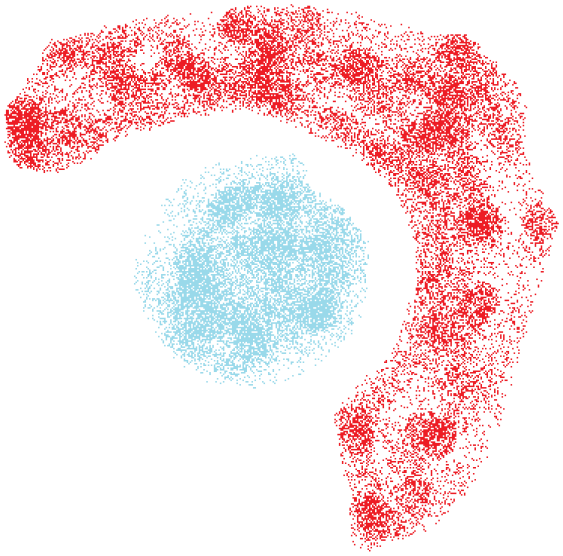




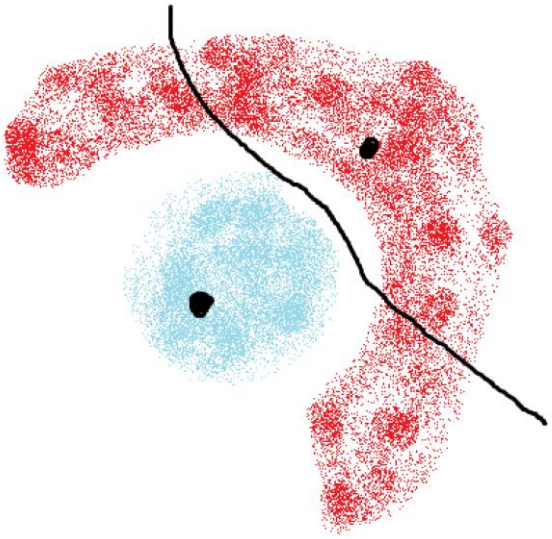
These centroids resemble their respective digits.

2.3

Example 1:



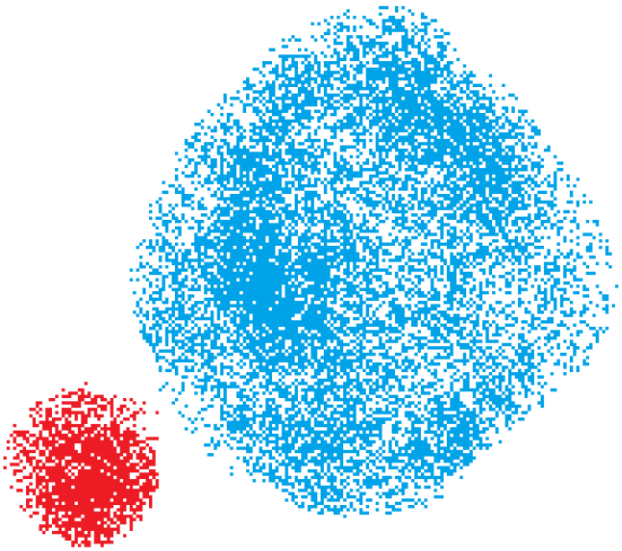
If we had a dataset that looked similar to this, with non spherical clusters, k-means could fail to identify the clusters correctly. To us this may be obvious, but a k-means clustering algorithm could find the centres to be:



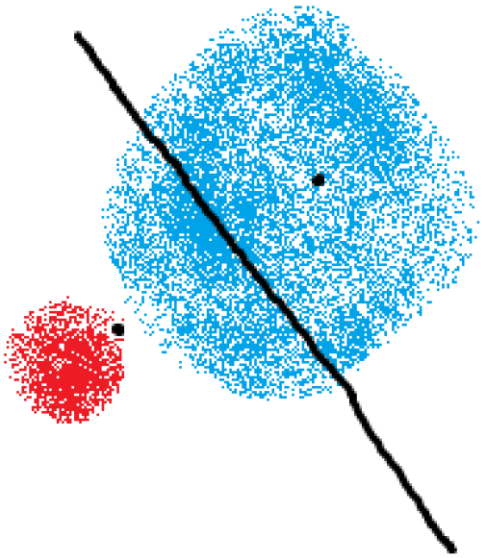
And clearly this is not correct, as the clusters are not defined by how close they are to the centre. We therefore require the clusters to be close to spherical for the k-means algorithm to be effective.

Example 2:

Even if the clusters are spherical, there can be limitations to the k-means algorithm's accuracy. For example, if the clusters look similar to this:



With close to spherical/circular clusters, then some of the points in the blue cluster are closer to the centre of the red cluster than the blue cluster. This means that the centroid corresponding to the red cluster will be drawn towards the centre of the blue cluster, as this is where the mean of the closest points are. Therefore, the clustering algorithm may generate the clusters to look something like this:



Which is clearly incorrect, which shows that another limitation of k-means is that the clusters should all be of similar size.

Question 3:

The code for this question can be found in decision_tree.py.

3.1

To find out which of the features was 'most important', I decided to calculate the information gain for each attribute. To do this, we use the formula:

Information Gain = Entropy(target) – $\sum$Entropy(target | a)

For each attribute a.

The target attribute is clearly most_present_age, so to calculate Entropy(target), we split the whole set of data by age, and calculate entropy with $\sum p_i \log_2 p_i$.

To calculate Entropy(target | a), we split each of the lists given by splitting the data by age by the attribute a. We then calculate the Entropy of each of these splits.

Because the data in each attribute a is continuous, as they are all numbers, we need to decide on a threshold that splits the list.

In order to find the best threshold, we sequentially try each attribute in the dataset as a threshold, and find the threshold which gives the lowest entropy.

After repeating this process, I printed a sorted list of each attribute, and its maximum information gain. The maximum information for an attribute is $\log_2 6 \approx 2.58$ bits.

```
[('vertical_density', 1.6137621001758917),
('Roads:diversity', 1.599741634858372),
('LandUse:Mix', 1.5838254116150619),
('buildings_age:diversity', 1.538278041611606),
('poisAreas:area_park', 1.5177677695117682),
('Roads:total', 1.483464612563918),
('Roads:number_intersections', 1.4782968853659202),
('Buildings:total', 1.4667369597456104),
('TrafficPoints:crossing', 1.4441990790786894),
('pois:diversity', 1.4340016476498136),
('poisAreas:area_pitch', 1.4241404547318182),
('pois:total', 1.36171351619631),
('ThirdPlaces:edt_count', 1.3325209889450416),
('buildings_age', 1.3218161657075629),
('ThirdPlaces:cv_count', 1.2945697550462725),
('ThirdPlaces:diversity', 1.2808252434932368),
('ThirdPlaces:out_count', 1.2579505124816246),
('ThirdPlaces:oa_count', 1.2560353432648614),
('Buildings:diversity', 1.2519948704207065),
('ThirdPlaces:total', 1.244099824535682)]
```

From this, we can see that 'vertical_density' is the most important attribute, followed by 'Roads:diversity' and 'LandUse:Mix'. The fact that vertical density is the most important makes sense in context; if the population is generally older in an area, then they do not tend to live on high floors, and sometimes even live in bungalows and similar buildings. Because

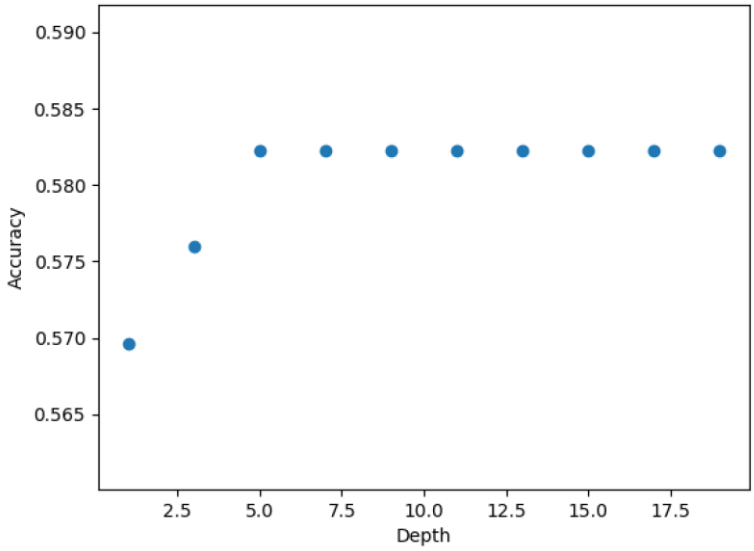
over 60 is also the most prominent age category in the dataset, this means that a high vertical density will separate a lot of the older people away, giving a lower entropy, and a higher information gain.

ThirdPlaces:total has the lowest information gain. However, the information gain is still around 1.24 bits, which is quite a lot of information. In our decision tree, this will be the bottom node, so we can observe the effect of removing this node from the tree in the next part.

3.2

In order to measure the accuracy of the data, we need to make a decision tree. Using this decision tree, we find that the accuracy of the prediction is around 58%.

In this data, the vast majority of entries are either under 18 or over 60, which means it is very rare that the decision tree predicts an age category other than these two.



In varying the depth, we see that the accuracy increases as the depth increases up to a depth of 5. This answers our previous question in question 3.1, in showing that only the top 5 most important attributes have an effect on the accuracy, and anything after that is irrelevant in the decision that the tree makes in the end.