Artificial Intelligence and Applications Coursework

1 Implement a heuristic search algorithm: A* and the 8-puzzle game.

1.1 How you would frame the 8-puzzle problem as a search problem?

Framing the 8-puzzle as a search problem is fairly simple. A graph can be constructed, where the nodes represent different states of the game and an arc between the nodes represents that one game state can be reached by making a legal move from the other. Each arc will have a weight of 1 so that when it is searched a minimal solution is found. This graph can then be searched from the start state until the goal state is reached, to obtain a set of moves to solve the problem. Another way this could be done is by constructing a tree from the start state in much the same way as before, where each node is a game state and the children of each node is a game state that can be reached from the game state by making a legal move. The tree could terminated at a certain depth where generating more nodes would be computationally infeasible or where a solution would definitely have been found. A breadth first search could then be run on the tree to find the shortest path.

1.2 Solve the 8-puzzle problem using A*

1.2.1 Briefly outline the A* algorithm

The A* search algorithm is a heuristic graph search algorithm. This means that when it is deciding which node to travel to next, it not only considers the the total length of arcs travelled, but also a heuristic function. For example if A* is being used for calculating the shortest route between two cities, the heuristic function may be the straight line Euclidean distance between the current position and the goal city. In this case the considered value at a node would be the total distance to travel to that position plus the euclidean distance to the goal.

1.2.2 Describe two admissible heuristic functions for the 8-puzzle problem

One heuristic function that could be used is the Hamming Distance, i.e. the number of tiles that are mismatched from the goal state. This is admissible as the number of moves needed to reach the goal state is at least the total number of mismatched tiles. This means that it is optimistically estimating how many moves left to reach the goal state, this means that the A* will be optimal.

Another admissible heuristic function is the Manhattan distance. This is the sum for all tiles of the total number of horizontal and vertical moves that would be needed to get a tile from its current position to its position in the goal state. For example to get a tile from (2,3) to (1,1) the tile needs to move 1 to the left and 2 down, so the distance for this tile is $2 + 1 = 3$ and the Manhattan distance for the game state would be the sum of these over all the tiles. This again is admissible as it underestimates the number of moves needed to reach the goal state. This heuristic should be better than the Hamming distance as it is less optimistic and should better reflect how far away you are from the goal state.

I have chosen these heuristics as opposed to other heuristics as they are simple to implement and are natural for this problem. The Hamming distance is also far more optimistic than Manhattan distance, so it will be interesting to see how much of a difference this makes when being used by A*.

1.2.3 Implement two versions of the A* algorithm

The code for this is provided in A-star question.py. Upon running the program will ask whether you want to use the default start and goal states and whether you want to use Manhattan or Hamming distance as your heuristic.

1.2.4 Briefly discuss and compare the results given by A*

As expected A* worked faster when the chosen heuristic function is Manhattan distance to when Hamming distance is used. This is because the Manhattan distance is a less optimistic heuristic than Hamming distance is. This is apparent by calculating the heuristics for the start state. The Manhattan distance of the start state is 18, whereas the Hamming distance is only 8. This is a drastic difference and affects the performance of A* greatly. By using the *timeit* library in python I ran the search on the start configuration ten times for each heuristic and calculated the average time to run. The result of this was that it took on average 3 seconds for the A* to reach the goal state using the Manhattan distance, while the average time to find the goal using Hamming distance was 444 seconds. To calculate this average I set the *timeit* function to run A* with each heuristic 10 times.

Ideally I would have used a larger sample size, however the run time of using the hamming distance made this infeasible. This shows how important it is to use a heuristic function that is representative of the work left to reach the goal, while still being admissible.

1.3 Bonus question

The code provided in 'A-star question.py' should work for any N-puzzle. It implements a check at the beginning to ensure that the goal state can be reached by the start state.

2 Unsupervised learning: k-means clustering

2.1 Describe k-means clustering and how it applies to the task of hand-written digit recognition.

K-Means Clustering is an unsupervised learning method of finding patterns in data. Given k, the algorithm will sort your training data into k clusters. You can then use the centres of these clusters to classify new data. There are two major elements to the K-Means algorithm, evaluating the clusters and evaluating the cluster means. To start off the algorithm, k elements in the datasets are randomly chosen to be the cluster centroids. The dataset is then sorted into the cluster with the centre it is closest to, once this is done the centroids are re-evaluated as the mean of the cluster. This is then repeated until either the maximum iterations is reached or the clusters stay the same. This can easily be applied to the problem of digit recognition. An image of a digit can be represented by an $n \times n$ array where each each value is a float representing either the colour or brightness of the corresponding pixel. This array can then be flattened into an n^2 dimensional vector. K-Means can then be applied to the set of these vectors, where the mean is the natural mean on vectors and distance is a norm of the difference between two vectors.

2.2 Using k-means

2.2.1 Analyse the data using k-means clustering

The code for this is in 'K-means question.py' and uses my own implementation of K-means.

2.2.2 Present and discuss the results of the k-means clustering on the dataset

To analyse the digits data set I ran my K-Means algorithm for $k = 1, \ldots, 15$ and plotted the sum of squared error for each of these. There was no obvious

"elbow" in this plot and it look akin to an e^{-x} graph, however there seemed to be a possible elbow in the 8 to 12 range, so I therefore decided to conduct some silhouette analysis for these values of k. These silhouette plots, shown in Figure 1, show that 8, 11 and 12 are not suitable numbers of clusters as the widths of the clusters are varied, implying that either a cluster encompasses multiple of the actual clusters or the natural clusters are split. This just leaves 9 and 10 as possible clusters. Of these the average silhouette score for

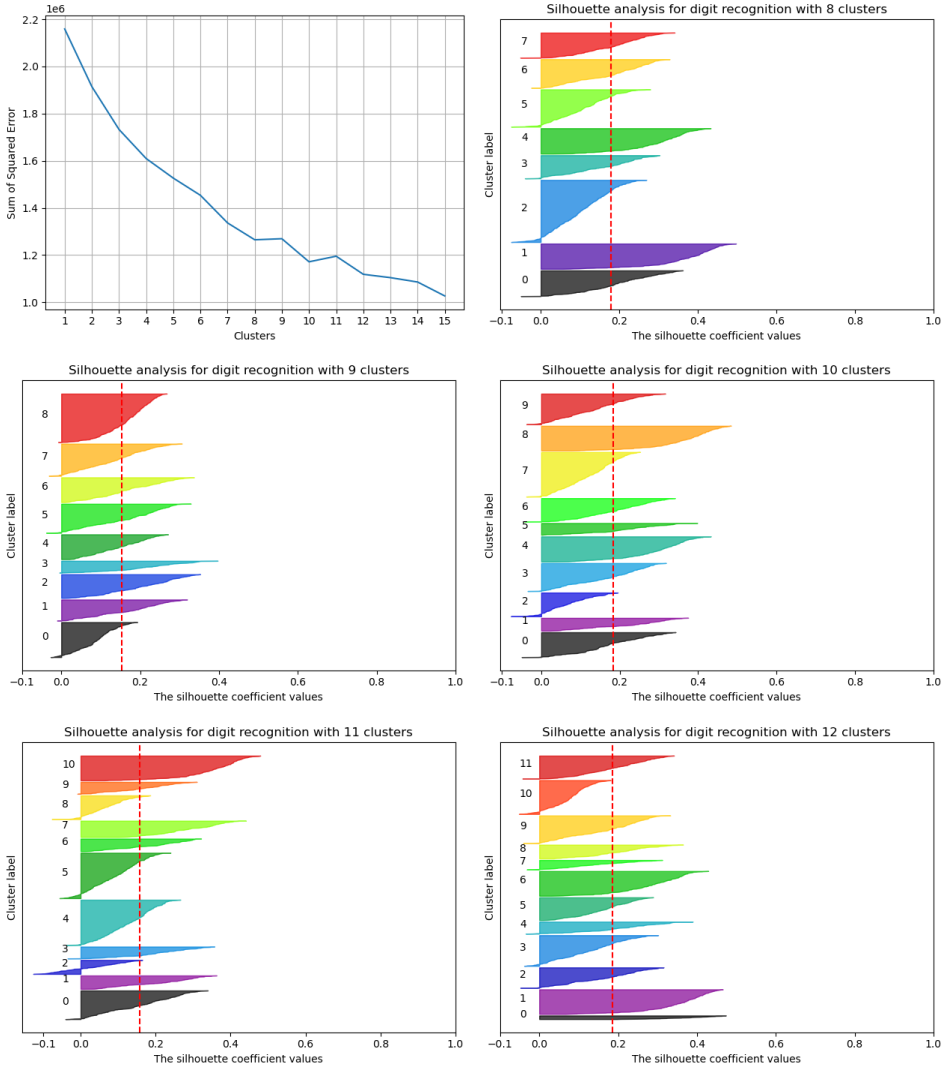

Figure 1

10 clusters was slightly higher, therefore this is the best number of clusters to use. This was to be expected as this dataset is labelled. The average silhouette score for all of the plots was quite low, which implies that the

data is quite close together despite the clusters, this can be seen by looking at the raw data as the middle of the images are quite dark while the outsides are light.

Since the data is labelled it is quite simple to calculate the accuracy of the clustering. To calculate the accuracy for each cluster I took the frequency of the most common label then divided it by the number of elements in the cluster. I then took the overall accuracy of the clustering by averaging all of the accuracies. To try to make the accuracy figure as accurate as possible I decided to run this 1000 times then take the average of this as the figure. This process gave the accuracy of my K-Means algorithm with 10 clusters as 79.17%. There are a few reasons that this figure is not higher. The main reason is that my K-Means algorithm is quite basic compared to other implementations such as the sklearn implementation and is likely not choosing the start centroids very well, and therefore finding local minimums that are not close to the global minimum for many of the times it is run. Another reason is that the data in this dataset is quite low resolution, meaning that elements may be quite similar even if they actually belong in different clusters. This makes it more difficult to pick out centroids so in some cases there will be two centroids that represent the same numbers and then a number that doesn't have a representative centroid. Another way to improve this would be to "clean up" the dataset to remove any outliers. This would help to make sure that centroids were more representative and increase the accuracy of the clustering.

2.3 Limitations of k-means

The K-Means algorithm assumes that any clusters in data are "spherical", this means that if there is a clear grouping in data but this in not in a circular cluster the K-Means algorithm will struggle to find it. For example in Figure 2 the data should clearly be clustered as the outer ring in one cluster and the inner circle as another, however K-Means is unable to see the data in this way as a point on the outer ring will be closer to points on the inner circle than another point on the outer ring but on the other side. If the outer ring was in one cluster then the centroid for this would be in the central circle and the cluster would then also include some of these points.

K-means can also only be applied to data with a natural mean and distance, i.e. numerical data or data that can be translated into a numerical form. For example if an attribute to the dataset is the season of the year, this can't be translated into a continuous value where a mean can be applied. A large drawback on K-Means is the effect that outliers in the data have on the effectiveness of the clustering. For example if there is a large outlier in the sample, it will be clustered with the nearest centroid then when the centroid is re-evaluated it will pull the centroid of the cluster towards it, potentially displacing data-points that did belong in the cluster.

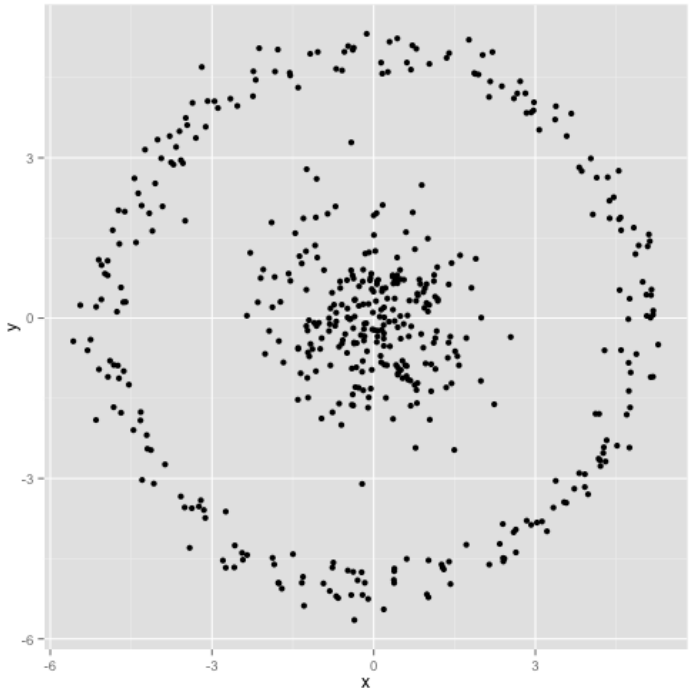

Figure 2: Credit David Robinson http://varianceexplained.org/r/kmeans-free-lunch/

2.4 Bonus question

For this question I decided to use the sklearn K-Means implementation as it is faster and better than mine. To analyse this dataset I first decided to only use attributes that all of the Pokémon had data for and were numerical. These were the against_type attributes,attack, base_egg_steps, base_happiness, base_total, capture_rate, defense, experience_growth, hp, sp_attack, sp_defense and speed. I also decided to remove the Pokémon Minior from the dataset as it had two possible values for capture_rate.

The plot in Figure 3 is what I found when using the elbow method to decide on a the number of clusters to use. I decided from this to use 6 clusters as this is where the plot flattens. Upon analysing these clusters I found that it was just grouping the data by the experience_growth. This is because this attribute is much larger than any of the others, in the order of 10^6 or 10^5, whereas the against_type attributes never get above 4. This meant that this attribute was greatly skewing the 'distance' between the vectors. To get round this I decided to normalise the data by making each value a fraction of the maximum of that attribute.

The elbow plot for the data after being normalised is shown in Figure 4. This however didn't show any clear "elbow", which suggested that there

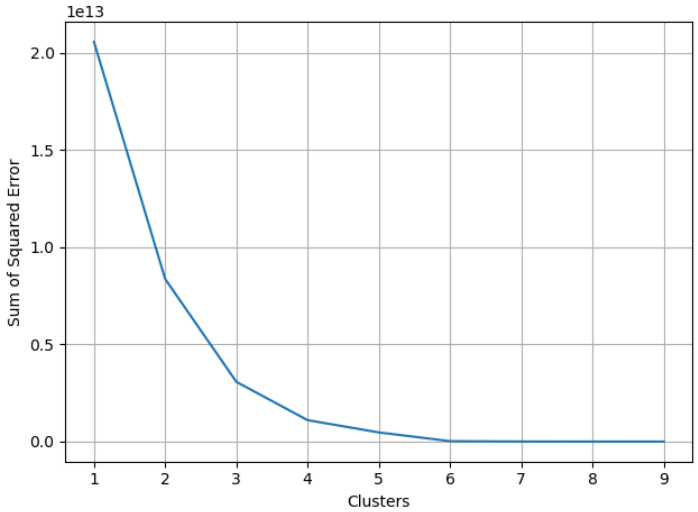

Figure 3

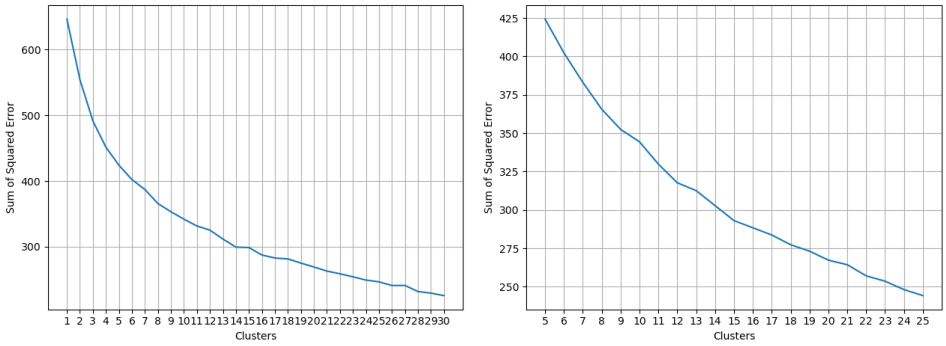

Figure 4

weren't any clear clusters in the data. After seeing this plot I decided to only use the against_type attributes, to see if this gave any clear clusters.

The plots in Figure 5 show the elbow plot when only using the against_type attributes. These show a much clearer "elbow" than before in the mid to late teens area. As there are 18 Pokémon types and the "elbow" was around this area, I decided to try 18 clusters. Upon analysing the clusters found that I the data had been clustered quite well into the types, for Pokémon with two types they were mostly on clusters that matched one of their types. As expected there were some anomalies, such as ice types largely being clustered with the normal types however this is only a small portion of the data. This shows that the type of a Pokémon can be found only by looking at what types that it is strong against.

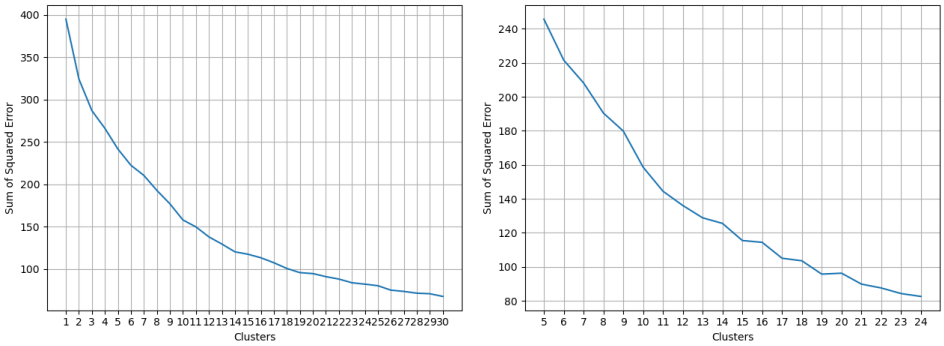
Sum of Squared Error
Clusters
Sum of Squared Error
Clusters

Figure 5