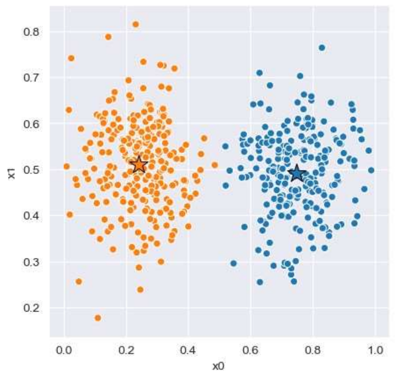
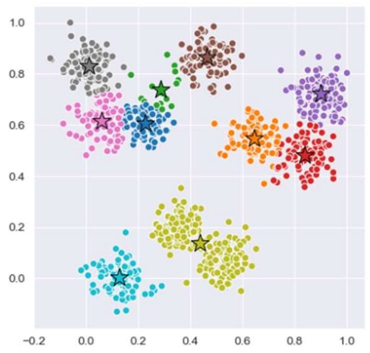
Assignment 1



For this assignment I have selected to implement K-means clustering and Decision tree.

To test the code, python unit test was used. These test files are called test.py and are in the corresponding folder.

K-means algorithm



The K-means algorithm begins with assigning location of k centroids. This is done according to some initialisation method. Then each datapoint is assigned to the closest centroid (Using Euclidean distance). Then for each centroid the mean of all the datapoints assigned to it, is set as its new position. Taking this mean and moving the centroid is then repeated until convergence.

K-means algorithm is well suited for unsupervised datasets and where there are obvious clusters when plotting the data. This can be everything from news articles by topic to stars in galaxies

Theoretically the K-means algorithm does not have inductive bias since it is an unsupervised algorithm and does not perform predictions. If this however was the case, then its inductive bias would be that its assignment would be to the closest centroid

To begin with I quickly realized it would be much more efficient to maintain two different data structures to store the datapoint-cluster assignment. I had a 1d list which stored the cluster assignment for each datapoint. Also, I had a 2d list where the 1-d list at index i, contained all the indexes for cluster i.

I originally implemented 3 different types of initialisation methods. These were Frogy, first K and random partition. However random partition was occasionally putting the centroids unreasonably far away, so that it had no points assigned to it. Therefore, it was removed. First K and Frogy performed equally well.

The second dataset was oddly scaled in the x0 direction. This made it look like the algorithm was behaving badly. It looked like there were datapoints which was assigned to a cluster further away than the closest one. I wrote a test case to check that each point was assigned to the closest one, which it was. To fix this issue both the x0, and the x1 was max normalised, and returned from the fit function, so that it could be plotted in a non-stretch manner

	Dataset 1	**Dataset 2**
Silhouette Score	0.672	0.483
Distortion	8.837	219.532

Decision tree

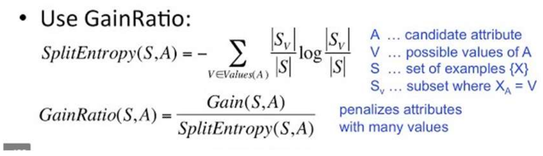
The goal of a decision tree is to generalize a set dataset. Predicting an attribute based on several discrete inputs. it does this by quantifying uncertainty, through entropy and information gain. It then uses this quantification to find ask questions such that the dataset can be generalised. These questions are asked on a specific column (or feature) and on a specific value in that column. Entropy is 0 when all values are equal and closer to 1 with more mixing (more uncertain).

To find the optimal question to ask, all possible questions are looped. For each possible question the dataset is split into true and false rows, then the information gain is calculated. This tells how much more certain the question splits the dataset (so that the true rows have one group, and the false another group). Information gain takes the current entropy and subtracts the weighted average of the entropy of the true and false rows. Where the weight is how many rows are in each of them.

This is then recursively repeated to form a tree structure. Where a new question is asked on the true/false rows. A leaf node with the predicted answer is generated when the entropy is 0, that is when the data contains only one group.

The algorithm can be modified to work on regression as well. Decision trees work well on discrete data which does not need to be updated. The structure of the tree changes a lot when the training data change a little.

The inductive bias of the ID-3 algorithm is that it prefers shorter trees over deeper trees. and high information gain attributes close to root is better than further away [1]. This is like Occam's razor, simpler decisions trees are better than bigger.



- Use GainRatio:

$$SplitEntropy(S,A) = -\sum_{V \in Values(A)} \frac{|S_V|}{|S|} \log \frac{|S_V|}{|S|}$$

A … candidate attribute
V … possible values of A
S … set of examples {X}
S_v … subset where $X_A = V$

$$GainRatio(S,A) = \frac{Gain(S,A)}{SplitEntropy(S,A)}$$

penalizes attributes with many values

The second dataset was harder to predict than the first dataset. The first dataset used the same dataset for training and testing, while the second dataset had separate datasets for training, validation, and testing. Also, the second dataset contained a lot more values. It is known that information gain favours columns with more possible values, this was compensated for using normalised information gain ratio (like in c.5-algorithm, see image). I implemented gain ratio slightly different; each value information gain is divided by its corresponding split entropy. This is related to the inductive bias because there my be a deeper more complex tree that gives better accuracy on the testset.

Accuracy (%)	Dataset 1	Dataset 2	Dataset 2 (normalised entropy)
Train	100%	100%	100%
Valid	n/a	74%	76%
Test	n/a	80%	83%

[1] https://web.cs.hacettepe.edu.tr/~ilyas/Courses/BIL712/lec02-DecisionTree.pdf

[2] https://www.youtube.com/watch?v=rb1jdBPKzDk