ECM2423 – Artificial Intelligence and Applications – Coursework

680033128

15/04/21

**Question 1.1**

The puzzle consists of a 3x3 board containing eight uniquely numbered tiles with zero as a placeholder for the blank space in the grid. This allows us to represent the states of the tiles by a numerical position with the top-left position being position 1 and the bottom right position 9.

Operations on the board would be moving tiles to fill the blank space left, right, up or down. We can set a goal test of reaching the goal state from the start state in the optimal number of moves. This is defined as the fewest position changes of the numbered tiles. After making a move the state of the board is either closer or further away from the goal state. In other words, starting from an initial node, we can develop an approach to solve the problem by finding a path to the goal node having the smallest cost. This path cost is equivalent to one unit per tile moved.

**Question 1.2.1**

A\* is an informed search algorithm that relies on weighted graphs to find the shortest path between a start and a goal node. It overcomes the shortfalls of greedy search by taking into account the cost of reaching the current node. To do this is uses an evaluation function to estimate the desirability of each node in the fringe. This allows the algorithm to expand the node that appears to be part of the cheapest path to the goal.

The other parts of the algorithm consist of g(n) which is the cost of moving from the start node to the current node. This is combined together with the heuristic function h(n) to give f(n) which is the estimated cost of the cheapest path to the goal state that goes through the current node.

A\* is complete and will always find a solution if one exists. We can also guarantee an optimal solution provided we are using an admissible heuristic which is one that never overestimates the cost to reach the goal. One major practical drawback is its space complexity O(b^m), as it stores all generated nodes in memory.

**Question 1.2.2**

An admissible heuristic never overestimates the cost of reaching the goal, it is considered optimistic.

Using an admissible heuristic will always result in an optimal solution.

1. Hamming distance (Number of misplaced tiles).

This is an admissible heuristic, as every tile that is out of position must be moved at least once to its goal position.

2. Manhattan distance (Taxicab distance of displaced tiles from their goal positions).

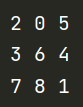
Traversing the grid horizontally and vertically we can find that for any tile not in its goal position it must be moved at least as many positions between its current state and its goal position. This means that the heuristic is admissible because we are considering a direct path between position whereas in reality tiles will likely be blocked from moving in a particular direction due to the unavailability of the blank space in one of the four possible directions.

I selected these heuristics because they have widely different levels of efficiency in determining the optimal path. The Manhattan distance is an example of a relatively efficient heuristic in contrast the Hamming distance is rather slow but does have the added benefit of being one of the simplest heuristics. Another consideration was to use Euclidean distance however this did not feel appropriate considering the restrictions in movement brought about by the sliding tiles as well as the fact that we are only need two dimensions for the puzzle.

**Question 1.2.3** -File: Question\_1/A\_Star.py

For both heuristics I have use the goal state given in the project briefing. However due to the inefficient nature the Hamming distance heuristic chose to use the start state shown below for both heuristics so that I can give a more meaningful comparison between the two.

Initial start state: Goal state:

 ****

The user informs the program of which heuristic function to use by entering either ‘1’ or ‘2’. At the top of the program, I have left the original start state suggested in the project briefing should you wish to uncomment this and test the Manhattan distance using this configuration.

If the puzzle is solved successfully then the user receives a printout of the key statistics an example of which is show below for the hamming distance heuristic function.

Puzzle Solved

In moves: 15

Depth: 14

Nodes visited: 5174

Total nodes generated: 8455

*The program will then proceed to print out the list of moves that follow the optimal solution.*

**Question 1.2.4**

The table below shows the performance of A\* using Manhattan compared to Hamming.

|  |  |  |
| --- | --- | --- |
| - | Hamming distance heuristic | Manhattan distance heuristic |
| Moves | 15 | 15 |
| Depth | 14 | 14 |
| Nodes visited | 5174 | 67 |
| Total nodes generated | 8455 | 113 |

We can see that the Manhattan distance heuristic does a better job of expending fewer nodes in order to find the solution with the least cost. This is because we can better prioritise which nodes are expected to lead to the optimum solution. Wherein Hamming distance, we are likely to get duplicate nodes with the same estimated cost, with Manhattan distance we reduce the chances of multiple nodes having the same f(n) value thereby leading to an improved queuing system. For any given tile, using Hamming distance the tile will either be displaced or not giving us an individual h(n) contribution of 0 or 1. By contrast when considering Manhattan distance contributions for any given tile, the maximum displacement can vary between 0 and 4. Therefore Manhattan distance leads to an improved queuing system by reduced the likelihood of two nodes having the same heuristic value.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Relative path** | Two moves prior to initial start state | One move prior to initial start state | Initial start state | One move after initial start state |
| **Start state** |  |  |  |  |
| **Time taken** | 1 minute, 38 seconds | 31 seconds | 15 seconds | 5 seconds |

The above table shows the impact of adding just a couple of additional moves to the optimal solution and how this affects time taken to evaluate the problem using Hamming distance.

For the four states shown there is only a single additional move separating each start state. Still, we can see that even changing from 15 to 17 moves in states 3 and 1, we increase the time taken to find the optimal solution by 1 minute, 23 seconds, a nearly sixfold jump.

**Question 1.3** -File: Question\_1/General\_A\_Star.py

The code for this question relies on the A\_Star.py module from questions 1.2.2 as well as the parity module found in the same directory. Run main method using General\_A\_Star.py.

This version of my program can take a user defined start and goal state and determine the optimal solution to get from one to the other. To do this we need to discuss inversions and parity which are key to determining which pairs of state are solvable and which are not.

For matrix √(n+1) x √(n+1) where n is the number of tiles, we have a 3x3 matrix with (n+1)! initial states of these ((n+1)!)/2 initial state are solvable.

In our 8-puzzle problem the number of possible arrangements (Permutations) is 9! which is equal to 362880 states. Only Half of the permutations 181440 are actually solvable. The other combinations of pairs of permutations are unsolvable.

Take for example the following:

puzzleInitial = [[2,8,3], [1,6,4], [7,0,5]] # Init. Inversion Count: 11 -> odd  
puzzleGoal = [[1,2,3], [8,0,4], [7,6,5]] # One. Inversion Count: 7 -> odd <- Solvable  
puzzleGoal = [[0,1,2], [3,4,5], [6,7,8]] # Two. Inversion Count: 0 -> even <- Not Solvable

If we consider each of the values within the initial state, we can tally the inversions. This is where a higher number proceeds a lower number moving from left to right across the matrix.

In this instance we can ignore the zero value as this is a placeholder for our blank tile.

Goal one:

8 proceeds the values 4,7,6,5 therefore we have 4 pairs of inversions.

7 proceeds the values 6,5 so add another 2 pairs of inversions.

6 proceeds the value 5 hence our total number of inversions is 4 + 2 + 1 giving 7.

Because we have an odd number of inversions, we can say that the goal state has odd parity. Therefore, it is only possible to reach the first goal state from an initial state that also has odd parity. Similarly for goal two we cannot reach this from the initial state because the initial state has odd parity, instead we would need a different initial state with even parity to solve this pair of states.

The Parity.py module provides the methods for ensuring that two states are solvable and the General\_A\_Star.py module validates the users input and initialises the start and goal states.

**Question 2.1**

Briefly outline the main concepts of the k-means clustering algorithm, and then describe how it could be applied to the task of hand-written digit recognition.

K-means in an unsupervised learning algorithm that can group collections of data points by discovering structures and similarities within unlabelled data. The aim is to partition the data into distinct non-overlapping groups referred to as clusters where each of the data points is assigned to only a single cluster.

Method:

Principle: Given centroids u(1).....u(k), assign each point to the best centroid.

We begin by creating k number of centroids which are randomly spread out. We can then assign data point to the centroid closest to it. This allows us to build-out our clusters. In the successive stages we recalculate each centroid based on the sum of the squared distance between all data points relating to a cluster. This process continues until there is minimal movement of the centroid over successive iterations. At this point we can say that the data points are grouped into tight clusters where net distance between a data points and its centroid is minimised.

In applying K-means to the task of recognising hand-written numerical digits we already know the number of variations within the dataset as we will end up with clusters for each numerical character. We can train the K-means algorithm to recognise the ten digits assigning a random centroid to each. From here we can continue to reduce the size of the clusters each iteration till we are left with tight groups of data which each represent a different digit.

**Question 2.2.1**

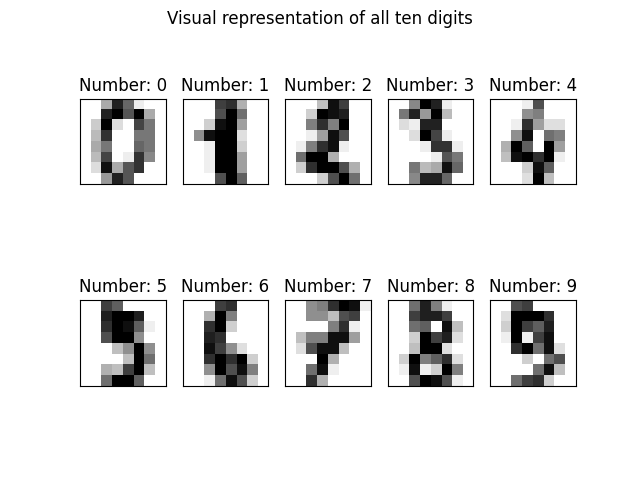
The code used to generate the graph for the discussion below can be found in the Question\_2 directory and includes the following files: k\_means.py, confusion\_matrix.py, visualise\_digits.py.

**Question 2.2.2**

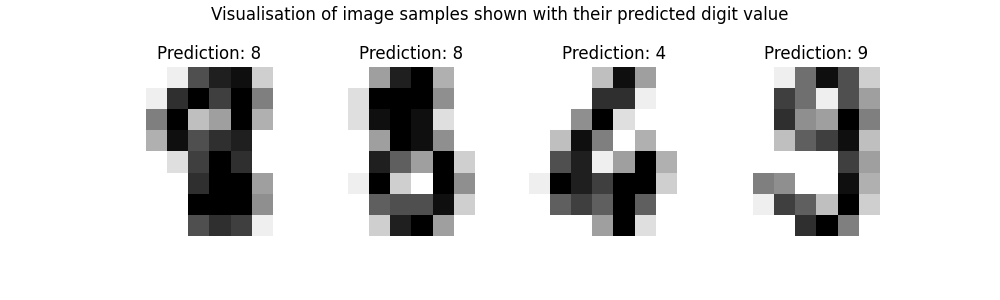
I have not implemented my own version of the K-means algorithm.

**Question 2.2.3**

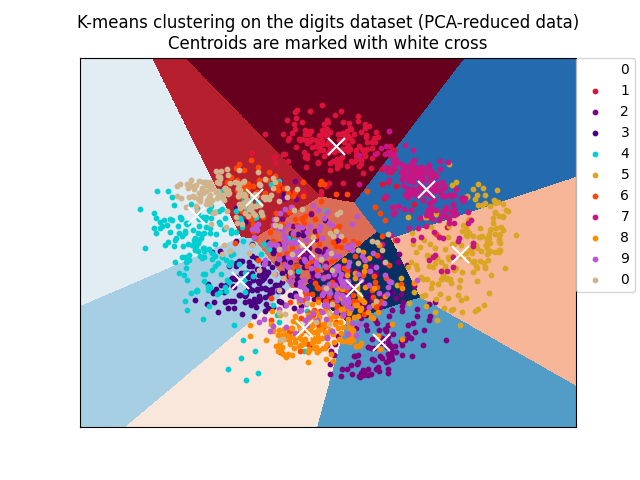
The sklearn handwriting recognition dataset consists of 8x8 pixel images of numerical digits. There are 1797 different sample images available, so I decided to split the set in-half and use half for training the half for testing. To apply a classifier on the data we can turn each image into an array of 64 grayscale values representing the 8x8 pixels. After fitting a support vector classifier on the training samples, we can then use the fitted classified to predict the value of the digits for the samples in the test subset.



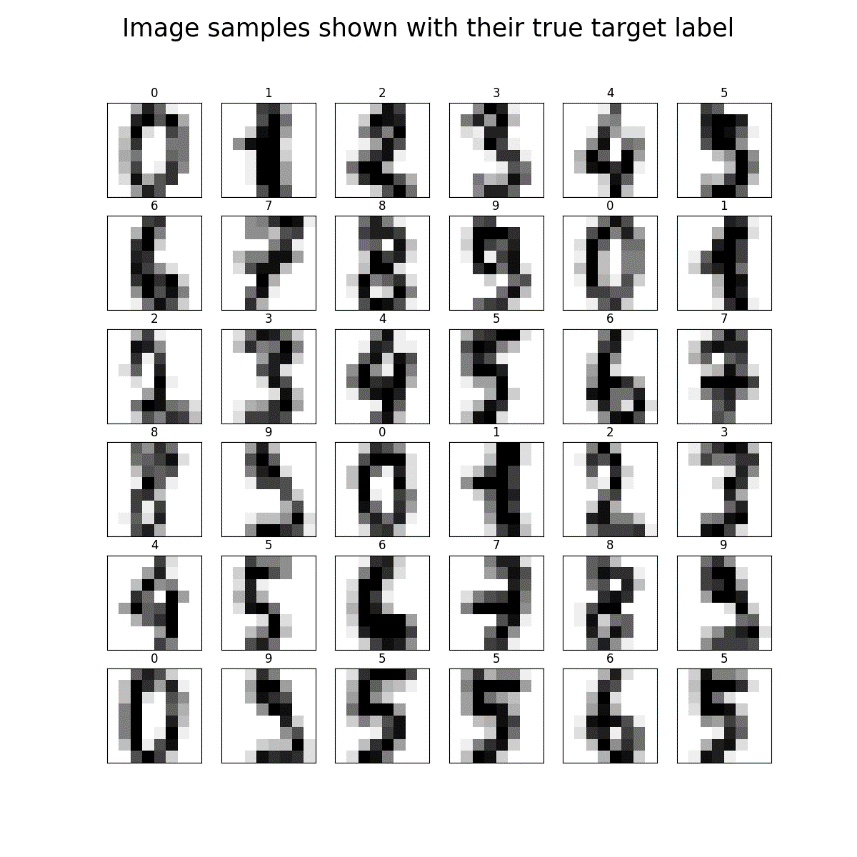
Here is an example of some of the predictions that the classifier might make.

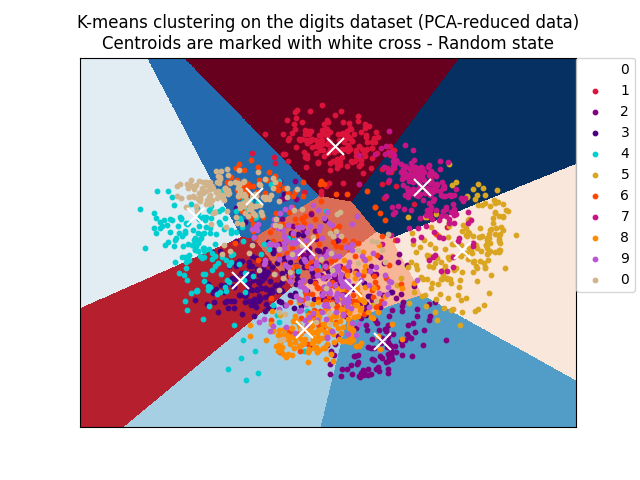


Here we can see the K-means clusters labelled with the target value for each digit in the set.

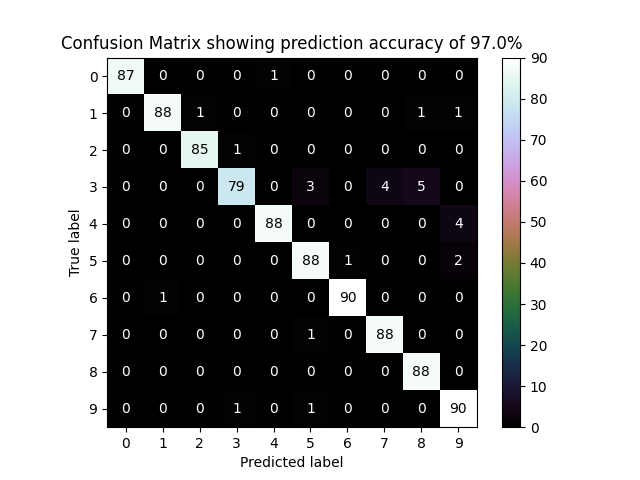


The background masking outlines the cut-off for each of the clusters. We can see that the centroids are fairly accurate for the majority of the clusters, however we can see that for the digits six and four we have fairly spread-out cluster. This could perhaps be improved by randomising the order in which the dataset is evaluated and increasing the number of times each image is categorised as the accuracy of the classifier improves over several iterations.



Below is an example where the data set is evaluated by the classifier in a random order. 

When running the K-means algorithm multiple times with the random\_state parameter set we can observer a slight shift in the position of the centroids. One explaination for this could be that the algorithm is giving a higher weighting to sample data that is categorised early on within the training process. We would have to look at the order of sample in the set to know for certain whether or not this has any effect on the accuracy of the model.

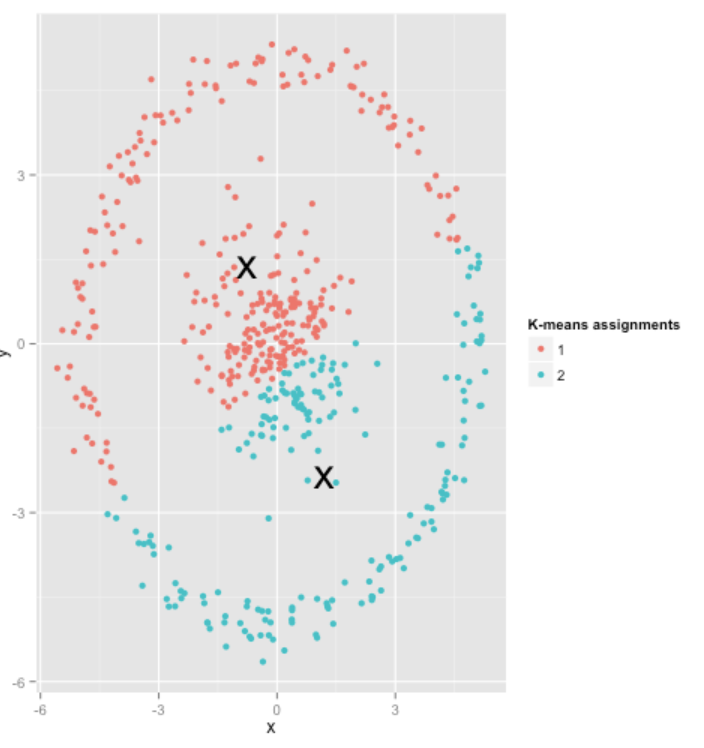
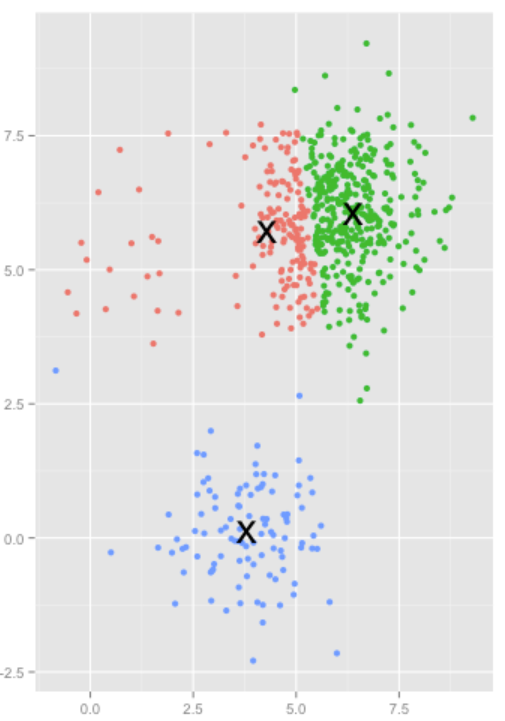


Above is a confusion matrix plot comparing the true values of the digits to the predictions.

This is a useful visualisation which can help identify target values that the algorithm commonly misidentifies. In the above chart we can see that the model incorrectly labelled the digit 3 as being the number 8 on five occasions. The algorithm on the whole still did a very good job of identifying the digit images as we achieved a prediction accuracy of 97%.

**Question 2.3**

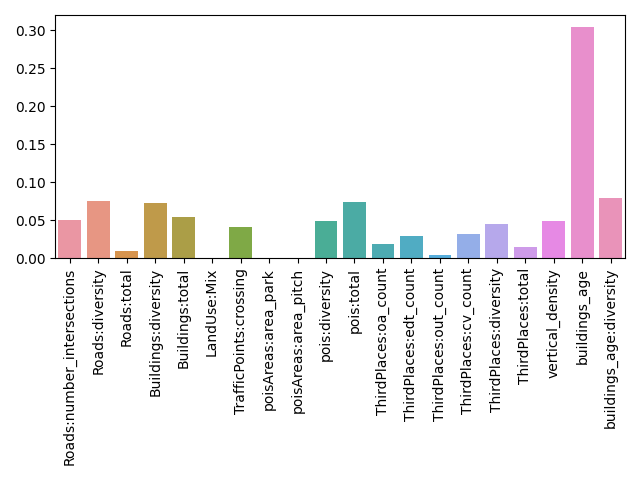
1. K-means may fail to work if the clusters form concentric rings. In this instance K-means is likely to highlight the centre cluster but will fail to find the outering ring. This is because the centroid will fall at the centre of the inner ring where it is equidistance from all data points. The algorithm could fail to acknowledge the existence of the second outer cluster or even worse attempts to split the cluster straight down the centre line as show in the diagram.

Example 1Example 2

2. Unevenly sized clusters - K-means expects a roughly equal split of data points within each cluster. If clusters are vastly different in size, then K-means will give a weighted bias towards the larger clusters and end up misplacing the centroid belonging to the smaller clusters.

**Question 3.1**

The code used to generate the graph for the whole of question three can be found in Question\_3/decision\_trees.py. This will produce two separate graphs and sets of statistics.

****

From the bar chart we can clearly distinguish that building\_age is a large contributing factor towards most present age. This has a weighting of 0.3 of the total combine influence of all the features being considered. The reason this factor is so massive could be because the younger generation would prefer to live in modern building. For example, perhaps they want to renovate their home which would be difficult with old building where there may be certain regulations forbidding the homeowner for making major changes to the property.

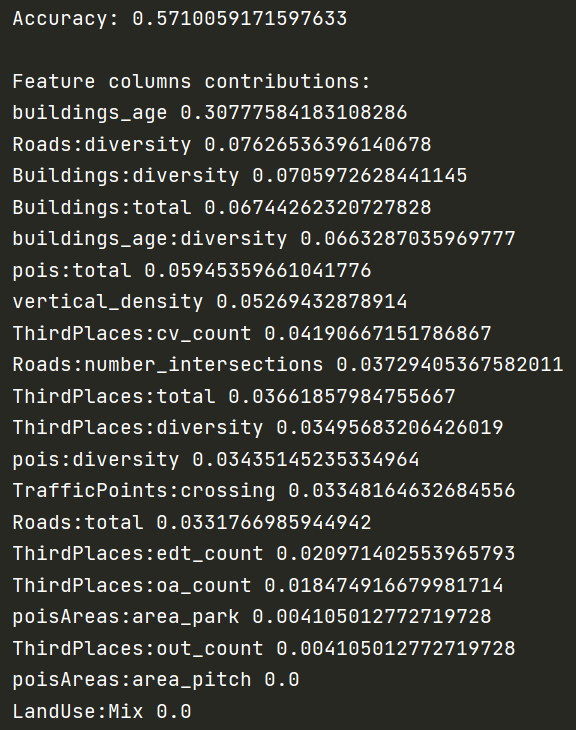
Aside from building roads:diversity is highlighted as another key factor. This is not that surprising because with older building we are likely to see narrower streets that may not be suitable for people cars making getting around the city more difficult. Any millennials living in one cell and needing to drive or take the bus to work may find they have difficulty. In contrast to cities there is a lot more potential for greater diversity in road connections to overcome the architectural limitation imposed as a result of when the city was built.

There are a number of other features that make up a similar proportion of the overall contribution. Among these are pois:total, building\_age:diversity and buildings:diversity which each account for approximately 0.07 of the total giving these features a relatively influence.

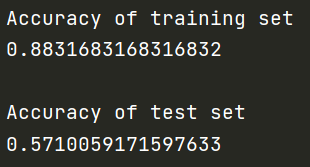
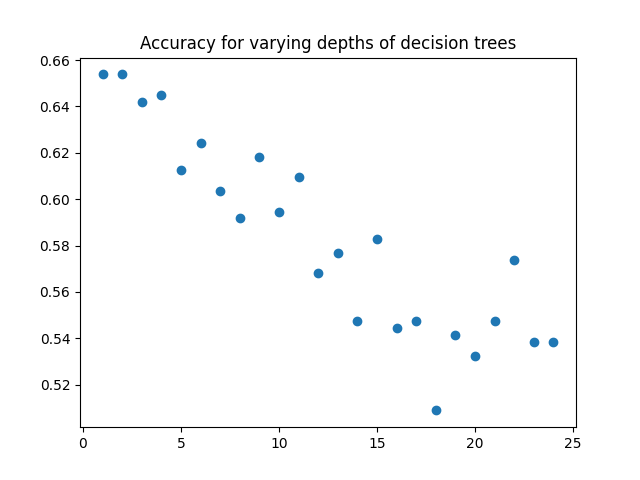
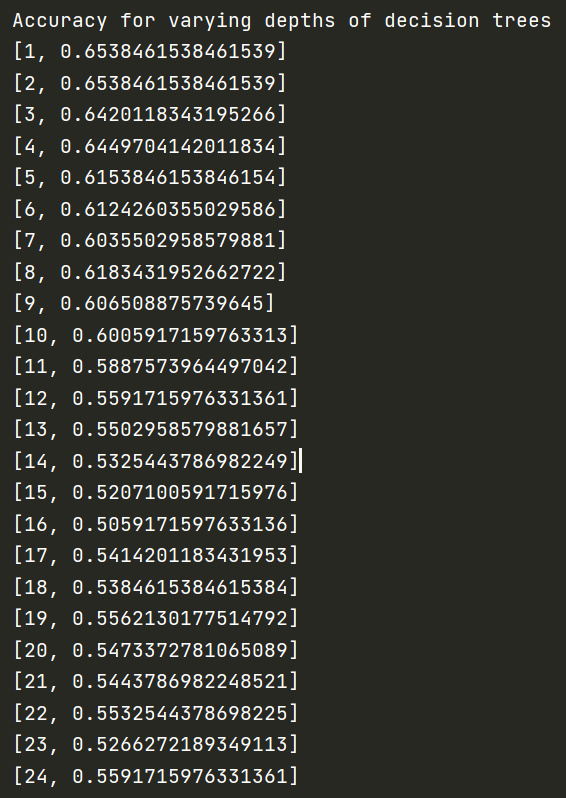
By contrast the following features have zero contribution toward the total, LandUse:Mix, and poisAreas:area\_pitch. These could therefore be omitted from the analysis along with several other feature that have a negligible effect such as poisAreas:area\_park and also ThirdPlaces:out\_count.

Perhaps of the lack of influence from the area\_park could relate to the diverse characters of people that use parts for recreation. Alternatively, there could simple be a smaller number of parks because the areas of the city are all highly build-up leaving little room for green space.

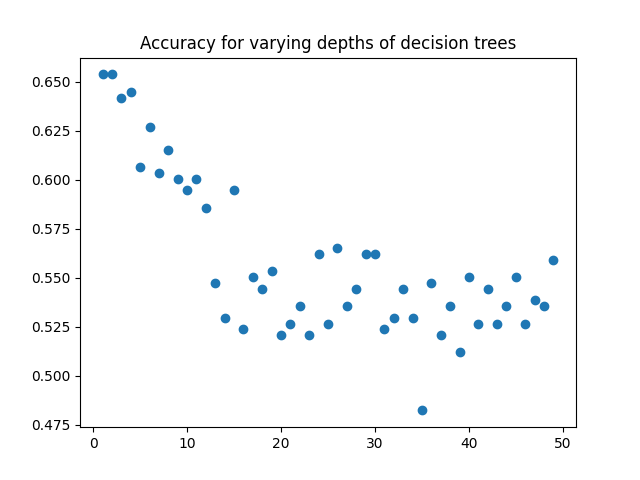
Contribution of features toward most present age of urban areas.

****

**Question 3.2**

As above, the code for this section is found in the same file Question\_3/decision\_trees.py****

We seem to have a fairly clear correlation between decision tree max depth and the accuracy of the model. The graph shows increase the depth the accuracy of the mode decreases. This is possible due to the fact that a deeper tree can fit more complicated functions. So, by increasing the tree depth we are able to the increase performance of the model on the training set. But, in doing so we end up overfitting the data, and generalisation performance it ends up suffering as the depth is further increases beyond a reasonable limit.



This is evident when we look at the above graph for a maximum tree depth up to fifty. As before, the first part of the graph follows a relatively constant slow. However, once we pass a depth of about 25 there is no longer any clear association between accuracy and tree depth.