Problem 1:

For the implementation of the Decision Tree using ID3 algorithm, I used some simple techniques.
I chose python as my language, as I find it apt for such programs.

The decision tree data structure was simply a dictionary of dictionaries. Here, you can enter the key, and get a result which can either be a decision tree or a string value.
I used the numpy array, as it gives me quick access to getting the column or row values from a matrix easily. This is required to count various values and labels in the ID3 algorithm.

The choose node part of the algorithm is implemented as a separate function, so that the ID3 heuristic can easily be swapped for any other heuristic if needed.

While I have tried to do a generalized decision tree with definable parameters, however, my accuracy was not coming out to be very good on the final result. To improve that, I made some changes, which may have led to the assumption, that the label is the 10th element of each line in the data set.

The id3Funct returns the node that the tree should be split on, by following the ID3 algorithm.

The tree can be viewed by simply printing the variable 'tree' from the main function.

Working :

The mainFunction calls 2 major functions.

1. makeDecisionTree, which makes the decision tree, and
2. testTree, which tests the tree for every record in the test set.

makeDecisionTree is a **recursive function**, which returns either a subtree or a label value.
The base conditions here are,

1. If all the data records or the valid attributes are consumed, the most frequent value is returned.
2. If only one label value is remaining in the current data subset, then that value is returned.

Otherwise,

1. it uses the ID3 heuristic to get the next node to split on
2. removes nodes that have been traversed from the list of valid attributes, and
3. for all the possible values of the current node, it creates a subtree, by calling itself, over a dataset consisting only of rows with the specific column values for current node

The id3 function typically tries to calculate the minimum Entropy feature. It does this by using the getEntropy() Function.

The getEntropy() function calculates the entropy of the given node by using an entropyFunction() to fetch the entropy over the labels for every value of the node, and multiplying the result with the probability of each value of the node.

The entropy function uses a subset of the data to count the number of positives and negatives, and creates the summed up log expression which is returned to the getEntropy() function.

All the files are read and concatenated and the tree returned by makeDecisionTree() function is tested on the test data.

Result:

My tree gives an accuracy of about 0.561, which is low, I went over the algorithm, and found I had made some mistakes. Removing those mistakes and putting other mistakes in raised the accuracy to 0.6, however, I further analyzed, and fixed more logical errors, which resulted in the accuracy going back to 0.561.

Problem 2:

The KNN problem was a fast and easier to implement problem.

Working

The first thing to do was implement a distance calculator. For this I have used the hamming distance between any 2 points.

The main function calls 2 functions. learnK() and implementKNN().
The file names of the dataset have been assumed to be in this format.

learnK() uses the function findKpoints() while changing the value of the k size, to find the nearest k points to a given point in the test data. This function then calculates the prediction, by using the findKResult() function to get the most common value in the K nearest neighbours.

learnK() iteratively changes k size, as well as files being tested, and finally after calculating accuracy for each k over multiple file sets, calculates and produces the average accuracy for each k, in the dictOfKAccs variable.

Result

learnK then returns the **K** with the **max accuracy**, which was **5**.
implementKNN() uses the K learnt by learnK(), and tests it on the testdata to calculate the **accuracy on test data**, which in this case was = **0.954**

K value	Accuracy
1	0.816
2	0.797
3	0.859
4	0.851
5	0.914