2802ICT Intelligent Systems
Assignment 2 Task 2 Report

By Nathan Cowan
S5143344

**Software Design:**

To run the KNN algorithm on the new dataset it had to be modified somewhat in order to function correctly.

KNN functions:

Many functions in this algorithm have similar names and purposes to ones in the DT algorithm and so are given the ‘k_’ prefix to distinguish them.

k_load() just like in the previous iteration takes a path to the input file and reads the data into a usable format. This time most of the data is taken as-is except the party which is converted to 1 for democrats and 2 for republicans before being moved to the end as to resemble the previous dataset.

euclidean_distance(), and sortkey() are unchanged from the previous implementation.

KNN() and k_accuracy() are both nearly identical to their task1 counterparts with the only modifications being that KNN() no longer has the commented option for alternative_distance() as it was not carried over, and k_accuracy() no longer prints any output.

DT functions:

load() as standard takes a path to the input file and reads the data into a readable format. The variables are converted from strings to ints and the party is moved to the end so to resemble the task 1 dataset so that the KNN could use the new dataset without any changes, however this did not end up being the case.

split_data() is exactly as it was in task 1 splitting the dataset by some ratio into training and testing.

check_entropy() takes some set of data and then calculates and returns the entropy between ‘democrats’ and ‘republicans’. There are some checks for edge cases such as if the dataset given is empty or consists entirely of one class or another and instead returns the hard values 1 or 0 respectively.

split_entropy() takes some set of data and the index of an attribute to split by, and then calculates the entropy for that attribute as a whole by calling check_entropy() on both the true and false subsets and multiplying them by their respective proportion of the original data.

information_gain() like above takes some set of data and the index of an attribute before simply returning the entropy of the original data- the entropy of the split.

next_attribute() takes some set of data and a list of remaining options. It loops through the options and returns the one that has the highest information gain as calculated using the function information_gain(). If no attribute gains any information suggesting that the data given is already of a single class this fact is returned instead.

next_node() takes some set of data and a list of remaining options. It gets the next best attribute to split by from next_attribute() and then returns the true set,false set, attribute split by, and remaining options after that split. If there is no attribute to split by and the data is already a class as indicated by next_attribute() then this is returned instead.

The tree is stored as an array in which each index is either a string of a class 'democrat'/'republican' for leaf nodes or an internal node in the form of a tuple which contains the attribute to decide by followed by the index of the node for true and the index of the node for false, where each of those nodes is either another internal node or a leaf node.

build_tree() takes the current tree, the index of the node it's working on, and some dataset, and a list of remaining options. If the options list is not given one will be constructed. It works by storing the returned value(s) of next_node(); if that value is a string representing that any further splits do not gain any information then the most numerous class in the remaining dataset is assigned to that index of the tree. If the next node should be a branch then a tuple as described above is assigned to that index with the children indexes pointing to two new nodes that are appended as 'None' before they are processed. In the processing loop if entropy is less than 0.01 then it is considered pure enough to be a leaf node, otherwise build_tree() recurses and goes through the whole process again to assign either a branch or a leaf value to the child nodes. This continues to recurse until options are depleted or all branches terminate. If options run out before there is a clear class then the most numerous is assigned. At the end the tree is returned. This is only relevant for the original function call as all recursion is in-place.

classifier() takes a complete tree, a sample to classify and an index which defaults to 0. If the tree node pointed to by index is a class then it's value is returned, otherwise it moves to the appropriate child node based on the value of the sample's attribute being split by that branching node and returns a recursion of that node. This has the effect of moving down the tree until a leaf node is found and then returning that back up as the classification.

Output functions:

The precision,recall and F1 values are calculated by the functions precision(), k_precision(), recall(), k_recall(), F1(), and k_F1() for democrats and republicans separately, that is a value where democrats are the positive class and one where republicans are the positive class.

The DT output functions all take the decision tree itself and the testing set as inputs.
The KNN output functions all take the training set, testing set and k value as inputs.

results() takes the built decision tree and testing set as inputs and then calls and prints precision, recall and F1() at once.
k_results() takes the training set, testing set and k value as inputs and then calls and prints k_precision, k_recall and K_F1() at once.

**Result:**

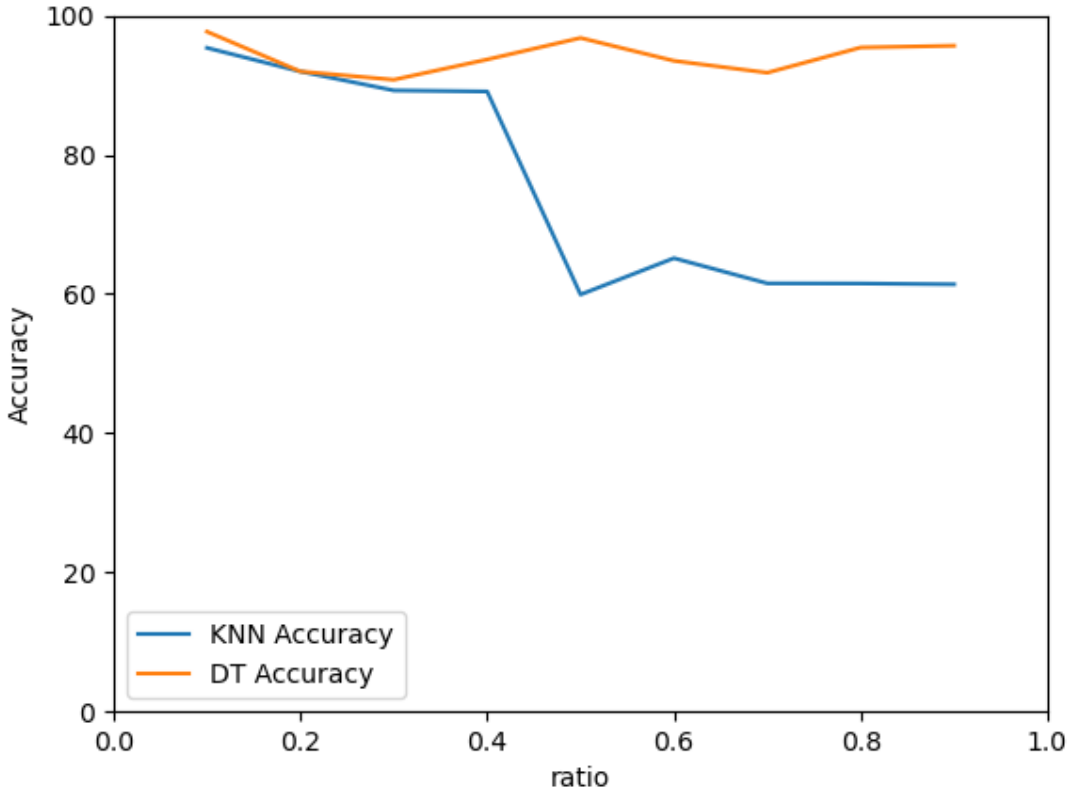


Figure 1: learning curve plot for both KNN and DT algorithms.

Figure 1 above shows how the accuracy of either classifier changes with diminishing training data. As can be seen, KNN is quite competitive when the training set takes up a clear majority of the data but swiftly falls to being ~1/5th better than random after that.

Having a testing set take up too much of the dataset increases the chance that what remains for training is no longer representative of the problem-space and any classifier built on it will become ineffective. KNN is simply more vulnerable to this happening than decision trees, however in some unlucky splits this can occur even for decision trees although this isn't visible in figure 1.

```
k precision = (1.0, 0.0)
k recall = (0.6129032258064516, 0)
k F1 = (0.76, 0)
precision = (0.9236641221374046, 0.9651162790697675)
recall = (0.9758064516129032, 0.8924731182795699)
F1 = (0.9490196078431373, 0.9273743016759777)
```

Figure 2: program output example.

Figure 2 simply shows an example of this program's output, specifically it shows the precision, recall and F1 values for both classes in both algorithms.

**Conclusions:**

K-Nearest Neighbors using a euclidean distance function is really not well suited for this dataset as euclidean distance is not a good metric in higher dimensions and more generally, neither is KNN. It is possible that KNN can become effective if variables are appropriately weighted but this is an effort all on its own.

Decision Trees are a surprisingly quick and effective means of generating a classifier with unknown training data.