# Intelligent Systems

## ASSIGNMENT 2 REPORT PART 2

Himath Ratnayake
S5209861 | GRIFFITH UNIVERSITY 2021

## Software Design – Part 2

### Functions:

| split_traintestdata | |
|---|---|
| **Description** | For this decision tree problem, it was decided to use the pandas library for its ease in data manipulation, as much of the decision tree problems center around operations that require the removal of column items, or splitting of them, which pandas excels at. This function will randomly split the training and testing data based on the split percentage passed in. |
| **Data Types/Structures** | The amount to split is calculated based on the number of data rows * percentage passed in.<br>The training set is initialized using the .sample pandas command to turn the data passed in into a pandas data frame, with the parameter "n" being used to initialize the amount of data points from the data that will be RANDOMLY picked and placed into the training set.<br>The testing data will be initialized by removing the all the data placed in the training data from the main data passed in using the .drop pandas function. |

| get_entropy | |
|---|---|
| **Description** | This function is used to calculate the entropy of a given set of data and a specific attribute. |
| **Data Types/Structures** | In the votes.csv file there are two possibilities: 1, which equates to a yes and 0 which equates to a no.<br>An entropy value is initialized as 0, and the np.unique function from the numpy library is used to count the number of times each unique value comes up in the input data frame, which is the column of the attribute we are currently looking at. It will then return the names of the two possible options into the classes variable ("0" and/or "1"), and the counts for each class into the counts variable.<br><br>Finally, a value "x" is initialized as being equal to the following:<br>$$x = \frac{number\ of\ "yes"\ OR\ "no"\ votes}{total\ amount\ of\ votes}$$<br><br>For each class, entropy is then found, following equation 9.3 below. The "sum of" portion of this equation is fulfilled through the for loop through the classes.<br>--- Equation (9.3)<br>NodeEntropy<br>$$\mathcal{I}_m = -\sum_{i=1}^{K} p_m^i \log_2 p_m^i$$<br>Note that $p^i_m$ is equal to x from the equation above.<br>`for i in range(len(classes)):`<br>`  x = counts[i]/np.sum(counts)`<br>`  entropy += -x * np.log2(x)` |

| get_highest_gain | |
|---|---|
| **Description** | This function is used to calculate the maximum information gain in a given data set. |
| **Data Types/Structures** | After a data set is passed in with the target attribute in question, the attributes column header is isolated with the data.columns.drop(attribute) function call from the pandas library. Below is an example of what this attribute_column data frame looks like.<br>`Index(['budget', 'physician', 'salvador', 'religious', 'satellite', 'aid', 'missile', 'immigration', 'synfuels', 'education', 'superfund', 'crime', 'duty_free_exports', 'eaa_rsa'], dtype='object')`<br>A dictionary is then initialized to store the information gains for each point of iteration in the attribute column. The goal is to find the split that will yield the highest information gain.<br><br>For this, the get_entropy function is used to first find the total entropy for the current "target" attribute in question (the attribute you're predicting for, which is party).<br><br>Then, a similar process that was done in the get_entropy function is done, but, this time for split entropy; for each attribute in the data, the number of yes and no votes is counted.<br>Then in the first iteration, a split occurs where all data equal in the current attribute is equal to "1". The number "x" is initialized as the number of data points in that attribute column which are equal to 1 or 0, divided by the total number of points in the data (i.e. number of "yes" or "no" votes divided by the total amount of votes). Then, the final split entropy is the "x" value multiplied by the entropy of the data that was split for the current iteration. This is then repeated for the data equal to "0".<br><br>$$I'_m = -\sum_{j=1}^{n} \frac{N_{mj}}{N_m} \sum_{i=1}^{K} p_{mj}^i \log_2 p_{mj}^i$$ --- Equation (9.8)<br><br>The information gains dictionary previously initialized is updated each time with the current attribute and its information gain, which is calculated as total entropy – split entropy.<br><br>$$Gain(A) = Info(D) - Info_A(D)$$ Information Gain (measures the expected reduction in entropy, or uncertainty)<br><br>Finally, the attribute which had the largest value of information gain after the split is extracted from the dictionary, and then returned back to the main build tree function. |

| build_tree | |
|---|---|
| **Description** | The build tree function is the main function that is responsible for the creation of the tree. |
| **Data Types/Structures** | Building the tree is a recursive function. For this implementation, the tree itself will stored in the form of a dictionary.<br>First, a root node is initialised, which will hold the decision tree's current root – this is what will be updated each time a new branch is added.<br>Then, the highest information gain value of the current attributes is gotten through the get_highest_gain function.<br><br>After this, the data is split at the point where information gain is maximised, using the split_data function, and the contents of this split are stored in the data_split dictionary. Then, the root dictionary is updated with this information.<br><br>For each outcome from the split node, the latest node is removed so that it is not reused again, and this updated set of data is stored in the tree_branch variable. Then, the purity of the current attribute for the newly updated data is found using the get_entropy function.<br>From here, there are two possible options:<br>• When the purity is 0, this means that all the outcomes are the same as each other (e.g. all 1s or all 0 votes). If this is the case, the root_node dictionary can be updated with this new pure branch.<br>• However, if all outcomes aren't the same, that means there is still potential for more branches to be created from this point onwards. From this point there are two more possible scenarios.<br>Firstly, it could be there are no more branches to divide on, which will happen when all the attributes are already divided upon (this will be when the length of the current number of attributes – 1 is equal to 0). At this point, the root node dictionary can be returned.<br>Alternatively, if there are still more attributes to be divided on, the build tree function is recursively called once again. For this new call, the current attribute's data points will be dropped (removed) column-wise (hence the axis must be specified as 1), as they have been evaluated already. |

| split_data | |
|---|---|
| **Description** | This function is responsible for splitting the data passed into the function on a given attribute. |
| **Data Types/Structures** | For the split data function, the new branch where the split will occur from is initialized as a dictionary. Then, the data is grouped by the current attribute, and the new branch dictionary is updated with these keys.<br><br>This newly split data is then returned. |

| Predict_decision | |
|---|---|
| **Description** | This function is used when testing to predict a decision on a test point. |
| **Data Types/Structures** | The tree that was created by the build_tree function is passed in along with a data point from the testing data.<br>Then, the dictionary in the form of a tree is traversed through recursive calls. At each recursive call, the given datapoint goes deeper into the dictionary branches (the sub dictionaries), and continues doing so until either a decision is reached, in which case the branch is returned, or the attribute is not found in which case None is returned. |

| find_accuracy | |
|---|---|
| **Description** | This function is used to find the precision, recall, accuracy and F1 values from the results of testing the tree on the test data. |
| **Data Types/Structures** | The tree built previous from the build_tree function as well as all the test data is passed into the function.<br>Then, four variables are initialized to 0: TP (true positive), TN (true negative), FP (false positive), FN (false negative).<br>Then, the test data is iterated through, and the prediction for a given test point in the iteration is stored in the "prediction" variable.<br>This prediction is then compared to the actual data. If the predicted and actual values are the same, this means that a "true" value was predicted.<br>For this data, "republican" was classed as positive, and "democrat" classed as negative.<br>Hence, if the actual and predicted values match, and the value was "republican", the true positive variable was incremented. However, if they match and the value was democrat, the true negative variable was incremented.<br>If the predicted and actual values do not match, then a "false" value was predicted; in this case, if the predicted value was republican, the false positive variable was incremented. Otherwise, the false negative value was incremented.<br>From here, it is simply calculated the accuracy, recall, precision and F1 values by substituting the found values into the equations.<br><br>The number of correct predictions is equal to: *TP + TN*<br>The total number of predictions is equal to: *TP + TN + FP + FN*<br>Accuracy is equal to: *(# of correct predictions / # of total predictions) * 100*<br>Precision is equal to: *TP / (TP + FP)*<br>Recall is equal to: *TP / (TP + FN)*<br>F1 is equal to: *2 * (Recall * Precision) / (Recall + Precision)*<br><br>The accuracy, precision, recall and f1 values are then returned, rounded to 2 decimal places. |

## Results

**Table 1: Average kNN vs Decision Tree Classifier Statistics Over 10 Trials**

| | kNN Algorithm | | Decision Tree | |
|---|---|---|---|---|
| **Average of Statistic** | **Training** | **Testing** | **Training** | **Testing** |
| **Accuracy (%)** | **94.1** | **93.2** | **99.34** | **94.7** |
| **Precision** | **0.95** | **0.93** | **0.99** | **0.94** |
| **Recall** | **0.93** | **0.91** | **0.99** | **0.90** |
| **F1 Score** | **0.93** | **0.92** | **0.99** | **0.92** |

With the same 70% training and 30% testing split in data, the kNN and decision tree algorithms were run 10 times and the average precision, accuracy, recall and F1 values were found as seen in table 1. The optimal values for k for the votes data set seemed to be around k = 13, with the normal Euclidean distance function. In general, it was found that the training and testing sets for the kNN algorithm had very comparable accuracies, with the testing set generally having lower accuracy when compared to the training set.

In the decision tree, the classification model was able to consistently predict above 98% accuracy for the training set, with an average of 99.34%. In comparison, the kNN was in a range closer to 92% to 96% for both the training and testing set; it had an average of 94.1% for the training set and 93.2% for the testing set. For the testing set of the decision tree, it was consistently lower accuracy than the training set, with an average testing set accuracy of 94.7%. This makes sense, as the classification model created in the decision tree is tailor made for the training set's data points and not the unseen testing set data.

Moving on to precision, the average precision of the kNN algorithm was 0.95 for the training set, and 0.93 for the testing set. For the decision tree, the average precision was 0.99 for the training set. However, the average precision for the testing set was 0.94, which is comparable to the kNN algorithm. Precision is the ratio of corrected predicted positive observations to the total predicted observations. Because in both data sets the precision is high, the rate of false positives being found can be considered low.

Recall is a value that shows the ratio of correctly predicted positive observations to all the observations in the class. Essentially, of all the republicans in the data set, how many were correctly labelled is what this value illustrates (the equations for this are elaborated upon in the software design section). In kNN, the training set and testing set once again had comparable values of 0.93 and 0.91 respectively. In comparison, the decision tree's training set had an average recall of 0.99. However, for the testing set in the decision tree, the recall was lower at 0.90. This means that for the training set, the decision tree performs well when correctly labelling party members based on their voting patterns in the training set, but is slightly less effective for unseen data.

Finally, the F1 score denotes the weighted average between precision and recall, meaning it takes both false positives and false negatives into account. In the case of the kNN algorithm, the F1 score averages over the 10 trials was 0.93 and 0.92 respectively for the training and testing set. For the decision tree, the F1 score was 0.99 for the training set meaning it was able to correctly predict in most cases without any false positives or false negatives. For the testing set in the decision tree however, it was an average F1 score of 0.92, which is comparable to kNN's F1 averages, and showed it was less effective on unseen data.

**kNN vs Decision Tree Classifier Learning Curves**

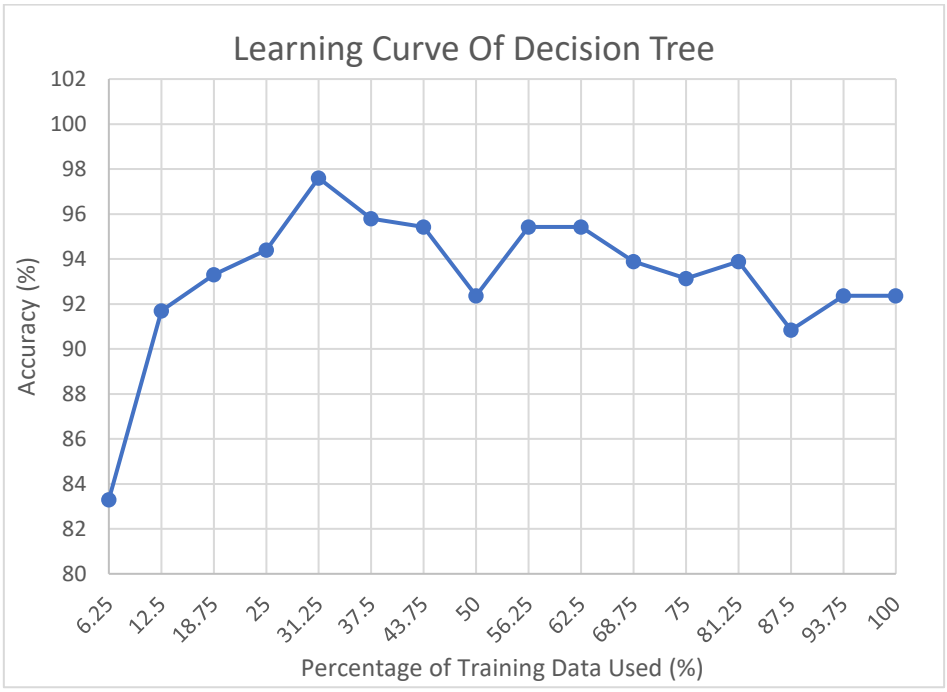


**Figure 1:** Learning curve of the decision tree

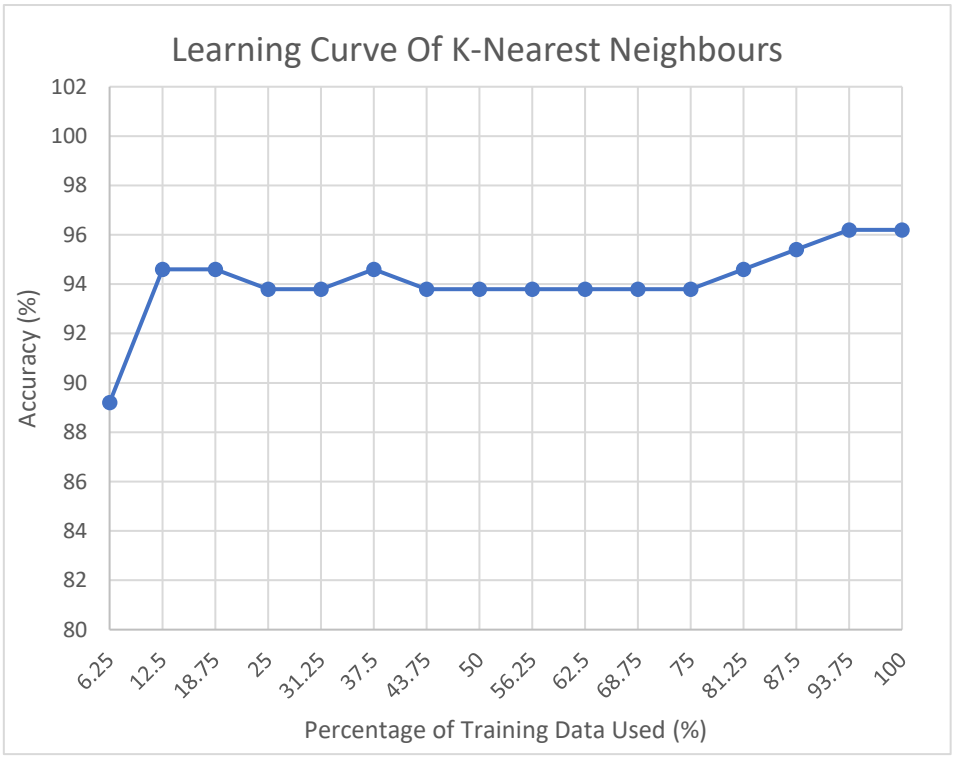


**Figure 2:** Learning curve of the kNN algorithm

A learning curve shows the accuracy of each algorithm over varying training sample sizes, as graphed in excel. The pseudocode and method for the building of this curve graph for both algorithms is demonstrated in the appendix.

For the decision tree, it was seen that the largest increase in accuracy was after the first increase in batch size, going from 83.3% accuracy in the first iteration to 91.7% in the second iteration. It can be seen there is some underfitting initially, which is to be expected given the small size of the training data for the first few iterations. Afterwards, the accuracy increased steadily for the remaining batches, approaching a maximum accuracy of 97.6, around when batch size equalled 95. However, from there onwards there were also some fluctuations where an iteration's accuracy was lower than the accuracy of the accuracy of the previous iteration. This could potentially be attributed to overfitting, which is more prone to occurring as training batch size increases due an increase in the model's complexity. In turn, this overfitting can mean that the decision tree relies too much on the training data and so performs worse on unseen testing data, which will mean that the model ends up performing more poorly – because of this, some methods could be utilised for future tests, such as reducing the training data size, or pre-pruning, which stops the tree from growing prematurely, so that the whole training set isn't classified. Post-pruning can also be done where some branches of the tree are removed after processing to reduce overfitting.

In comparison, the kNN algorithm's learning curve was able to achieve a higher accuracy % by the first iteration (89.2%), when compared to the decision tree's first iteration accuracy of 83.3%. However, the kNN algorithm's curve also plateaued much faster with a lower maximum accuracy. This means that compared to kNN, the decision tree was on average more accurate for larger data sets. This makes sense, as decision trees are better when the set of values in the training data is large. Compared to the "lazy learning" of kNN where it directly learns from training instances without creating a classification model, the decision tree is able to build a classification model on the training data set and accurately classify unseen data from the testing set. The strengths of each algorithm are summarised with the following table:

| kNN | Decision Tree |
|---|---|
| **Advantages:**<br>Utilizes a "lazy learning" approach where the algorithm learns directly from training instances without creating a classification model, which can be convenient. In kNN, it bases the prediction on similarity of the attributes. | **Advantages:**<br>In contrast, decision trees can take a brand new input and rapidly classify it; generally, decision trees are faster than kNN as they generate a classification model that can be reused for many unseen inputs.<br>A distance metric does not need to be chosen either as the splits are based on entropy, which is a value that is intrinsically within each data point. |
| **Disadvantages:**<br>kNN tends to be slower than Decision Trees, due to having to execute learning on the spot and give a result in real time.<br><br>Accuracy is dependent on a distance metric, so an effective one must be utilised.<br><br>Can be computationally intensive, especially for data sets with many features that need to be passed into a distance function. With large data sets, storage can also become challenging and inefficient, but for the iris dataset this was not an issue. | **Disadvantages:**<br>For a classification model (or tree) that already exists, it is difficult to update it with new attributes and a brand new tree needs to be recreated.<br><br>The building of the classification model through recursion generally takes much longer than kNN, but once it's built it is able to make predictions/classifications faster as it is a simple traversal down the tree.<br><br>Prone to overfitting, especially if too much training data is used in classification model. |

## Conclusions

It was found that both algorithms were effective at predicting and classifying data using supervised learning techniques. Ultimately, both algorithms have their pros and cons as described in the table above, with decision trees being generally slower at creating a model but faster at predicting once the classification model is created. Decision trees can benefit more from a larger training sample size when compared with the kNN algorithm and can predict from the data used in making the model (the training set) more accurately due to basing their classification model on said data. However, for unseen data, the accuracy can in turn be lower, as with too much of the training data being used, the model is prone to overfitting on the training data. Therefore, strategies such as pre or post pruning could be utilised in future when building the tree. In contrast kNN can predict without a creation of a classification model, and can be a better predictor when the training sample size is smaller, but can be much more computationally/memory intensive for larger datasets.

## Appendix: Building The Learning Curves For kNN and Decision Tree

For the purposes of this assignment, the training data size was 304 samples, which is equal to 70% of the total 435 data points. This training data was divided into 16 batches, with each iteration of running the data increasing batch size by 1/16. 304 divided by 16 yields an increase in batch size at each iteration of 19 (e.g. i = 1, n = 19, i = 2, n = 38, and so on).

**The pseudocode is as follows:**

Initialise batch size to 19

Iterate a variable i from 1 to 16:

At each iteration, initialise a list called “batch”, which has the elements from index 0 of the full training array (the start) until 19*i

Then, find the accuracy of running either the kNN or decision tree algorithm with this batch size.

- For the decision tree, a new tree is built with the current batch, and then this newly built tree is tested against the testing set.
- For kNN, this new batch is passed in as the training set, along with the testing set and k value

- At each iteration, the accuracy found from using the current batch size is appended to an accuracies array, and also written to an output file
- The results from this output file are copied and pasted into an excel document for efficient graphing

## References

Research on how to represent a decision tree (using dictionary):

- https://stackoverflow.com/questions/61831953/get-a-decision-tree-in-a-dictionary
- https://stackoverflow.com/questions/13688410/dictionary-object-to-decision-tree-in-pydot
- https://stackoverflow.com/questions/60422346/traverse-decision-tree-based-on-values-iteratively-going-into-sub-dictionaries

Pandas framework documentation used for various functions:

- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.idxmax.html
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html?highlight=drop#pandas.DataFrame.drop
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sample.html?highlight=sample#pandas.DataFrame.sample
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html?highlight=groupby#pandas.DataFrame.groupby
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.value_counts.html?highlight=value_counts#pandas.DataFrame.value_counts