CT4101 Assignment 2

Name: Aideen McLoughlin	Student ID: 17346123	Class: 4th year CS
Name: Louise Kilheeney	Student ID: 16100463	Class: 4th year CS

Description of Algorithm

For this assignment, we chose to develop an implementation of the C4.5 Algorithm.
Our algorithm starts by gathering the user input. The user must type the filepath of the file from which to fetch the data, and the train to test split fraction, selected from one of 3 options - ⅔, ⅓ or ⅓. i.e. if a user picks ⅔ then ⅔ is training and ⅓ is testing. If the file path to the data is wrong, an error window is loaded which will prompt the user to try again. Once a valid file path has been passed, the gather_data function returns the file data, and the split fraction. At this point, createTreeWhileShowingLoadingScreen is called, this function uses python multiprocessing to render an animation of a loading screen while the tree is being built, so that the user can know that there have been no errors, and the tree is building. At the same time as this loading animation function, renderLoadingWindow, is called, the createTree function is called. createTreetakes the file data and split value. createTree calls the function split_data_training_testing, which splits the data into training and testing data in the ratio of the split value. The function then saves these datasets so that the Weka implementation can run with the same data, then the datasets of train_data and test_data are returned to createTree.
Once the training and testing data has been returned, the buildTree function is called. buildTree takes the training data and the list of attributes as input.
Build_tree allows for 3 bases cases

1. Check that all data is not in the same class and return a failure leaf node fail if it is
2. Check that the training data set is not empty and return a failure leaf node fail if it is
3. Otherwise, Find the attribute with the highest information gain and return that attribute
    a. find_best_attribute is called, which takes the data and calls spilt_into_subsets which splits the data in half around a threshold value
    b. Then information gain is called, which passes the subset to the entropy function and uses the returned value to calculate the information gain of the attribute. The information gain value is then returned.
4. If the best_attribute function returns an empty string, then there are no more attributes to analyse, and a node can be created representing the majority class from the data.
5. Otherwise, a new node can be created for the best attribute, and the dataset is then split in 2 at the threshold value for that attribute. Those values are then appended as child nodes to the new best attribute node by recursively calling the createTree function, by calling build tree with in the subset and the unused attribute
6. Return the root node, which will return have connections to all the child nodes through the recursive calling of createTree

Once the tree has been created, the root_node is passed to Print_tree, which uses a Graphviz Digraph to visualise the tree and save it as a png so it can be displayed in the GUI.

The Accuracy of the tree is calculated by passing the root_node and the testing data from the tree to the test_tree function. The test_tree function removes the target column from the dataset (style) and stores those values so they can be used in evaluation. The test_tree function then calls the test_data function, which loops through all the test data values, and calls test_lr for each of them. Test_lr takes a row of data and the root node of the tree. It starts at the root node of the tree, and works its way down to the relevant predicted leaf node for the row data passed to it. The leaf node reached in the test_lr function is the predicted target for the row. Once all test rows have been evaluated, the test_data function returns a list of all the predictions, and then loops through them alongside the actual values to calculate the accuracy of the tree. Test_data then saves the test output to a file and returns the accuracy of the test.

The tree is then built using the Python Weka Wrapper module. The Weka C4.5 model is trained with the training dataset which was saved as a file in the training_testing_split function, and tested using the test data which was saved in that same function. The tree is then rendered as a png and saved, and the accuracy of the tree of the testing data is calculated. The results of the tests are stored in a csv file as was done in the python implementation.

The time it took to build the tree using the python code (built from scratch) and the weka code are formatted nicely, and then the final pop-up window will render, showing both the python tree and the weka tree, along with the accuracies of both trees built from the same data set on the same testing data set, and the time it took for each implementation to build the tree.

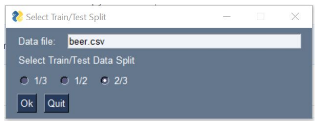
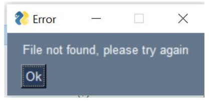
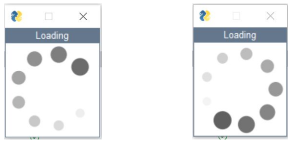
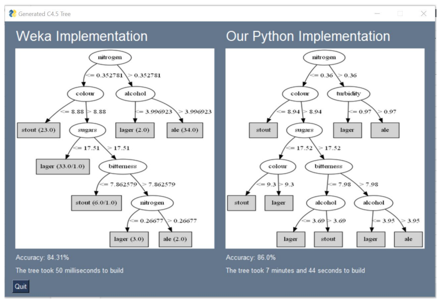
Design Decisions

Algorithm Design

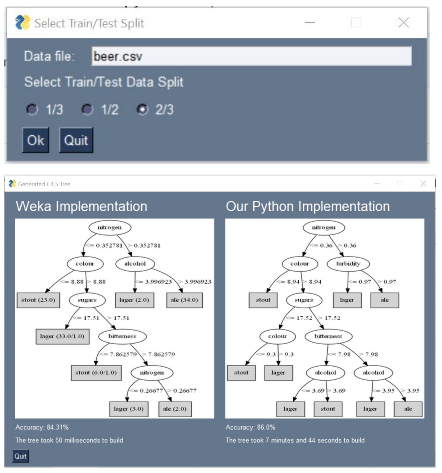
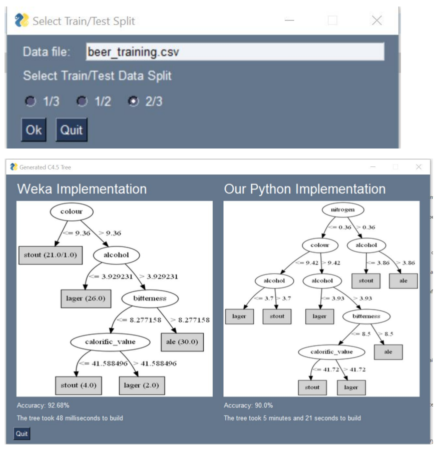
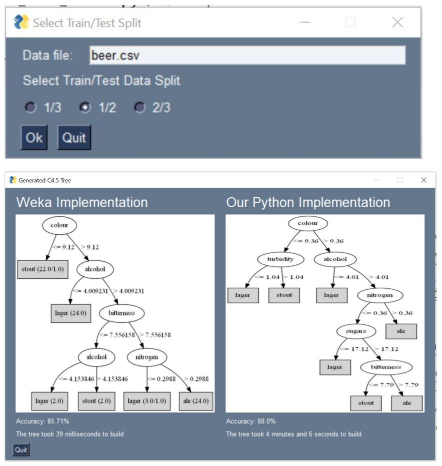
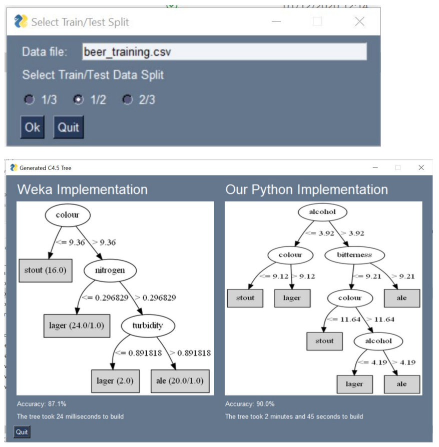
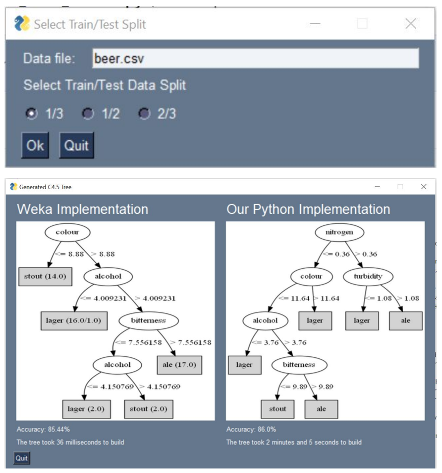
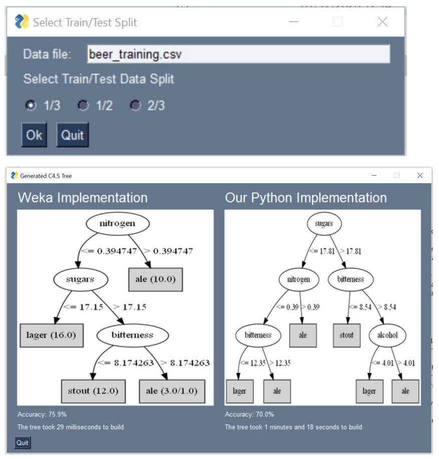
- We chose to use Python for this assignment as python facilitates the complex data operations required by the C4.5 algorithm , while still allowing for relatively easy implementation of elements such as the GUI.
- We chose to develop an GUI (A User Interface) for our algorithm implementation to make our tree building and testing clearer and easier to use
- We chose to input and process all of our data in CSV format as it made it easier for us to use and store.
- We chose to style the graphviz graph for our tree similarly to the weka tree for the sake of consistency, so that any graphical differences between the trees would be minimal
- We chose the split value options of ⅓, ½ and ⅔ to use in our GUI as they are all reasonable enough to allow for a nicely representative test and train data set, while being different enough to show how the split function can change the overall tree outcome.

Interface and Graphics Design

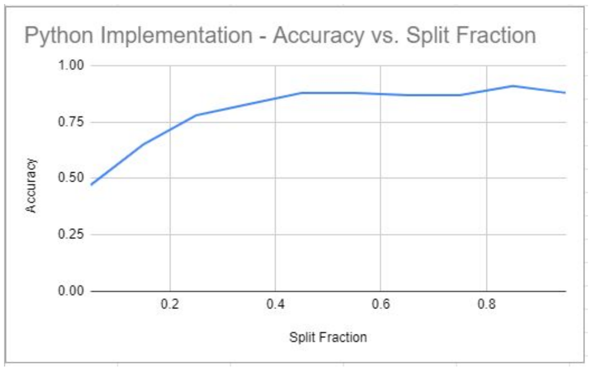
Our implementation of the C4.5 Algorithm runs with a simple Graphical User Interface built using the PySimpleGUI python library.

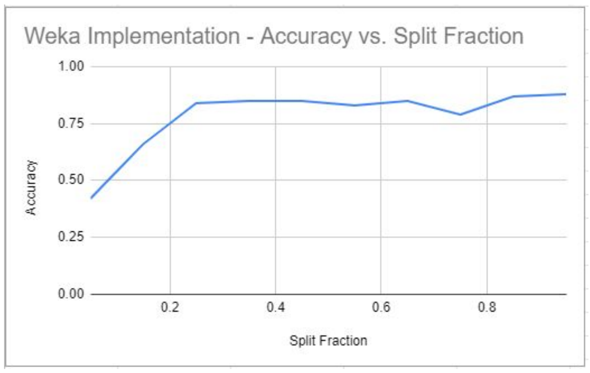
Action	Response	User Interface
Run the program	A pop up window appears, prompting the user to enter the data file location, and the split between training and testing data to use for that data	 Select Train/Test Split Data file: beer.csv Select Train/Test Data Split 1/3 1/2 [✓ 2/3]
Enter invalid data file location	Popup appears to notify you that an invalid file path was entered. Once User clicks OK, the initial window will pop up again so that the user can enter valid data	 Error File not found, please try again
Valid inputs in first popup, User clicks 'OK'	The program will begin building the C4.5 tree in the background. While this happens, a loading gif will be animated on the screen to show that the program is still working and not frozen.	
Once the tree has built	The loading popup will close, and a new window will pop up to show the Weka tree and the python tree built by our code.	 Generated C4.5 Tree Weka Implementation nitrogen <= 0.352781 > 0.352781 colour alcohol <= 8.88 > 8.88 <= 3.996923 > 3.996923 stout (23.0) sugars lager (2.0) ale (34.0) <= 17.51 > 17.51 lager (33.0/1.0) bitterness <= 7.862579 > 7.862579 stout (6.0/1.0) nitrogen <= 0.26677 > 0.26677 lager (3.0) ale (2.0) Accuracy: 84.31% The tree took 50 milliseconds to build Our Python Implementation nitrogen <= 0.36 > 0.36 colour turbidity <= 8.94 > 8.94 <= 0.97 > 0.97 stout sugars lager ale <= 17.52 > 17.52 colour bitterness <= 9.3 > 9.3 <= 7.98 > 7.98 stout lager alcohol alcohol <= 3.69 > 3.69 <= 3.95 > 3.95 lager stout lager ale Accuracy: 86.0% The tree took 7 minutes and 44 seconds to build

Tests and Results

Data	Output	Data	
Dataset: **beer.csv** Train/test split: **2/3** Accuracy: **86%** Weka Accuracy: **88.24%**	 Select Train/Test Split Data file: beer.csv Select Train/Test Data Split 1/3 1/2 [✓ 2/3] Ok Quit Generated C4.5 Tree Weka Implementation Accuracy: 84.31% The tree took 50 milliseconds to build Our Python Implementation Accuracy: 86.0% The tree took 7 minutes and 44 seconds to build Quit	Dataset: **beer_training.csv** Train/test split: **2/3** Accuracy: **90%** Weka Accuracy: **85.37%**	 Select Train/Test Split Data file: beer_training.csv Select Train/Test Data Split 1/3 1/2 [✓ 2/3] Ok Quit Generated C4.5 Tree Weka Implementation Accuracy: 92.68% The tree took 48 milliseconds to build Our Python Implementation Accuracy: 90.0% The tree took 5 minutes and 21 seconds to build Quit
Dataset: **beer.csv** Train/test split: **1/2** Accuracy: **88%** Weka Accuracy: **87.01%**	 Select Train/Test Split Data file: beer.csv Select Train/Test Data Split 1/3 [✓ 1/2] 2/3 Ok Quit Generated C4.5 Tree Weka Implementation Accuracy: 85.71% The tree took 39 milliseconds to build Our Python Implementation Accuracy: 88.0% The tree took 4 minutes and 6 seconds to build Quit	Dataset: **beer_training.csv** Train/test split: **1/2** Accuracy: **90%** Weka Accuracy: **87.1%**	 Select Train/Test Split Data file: beer_training.csv Select Train/Test Data Split 1/3 [✓ 1/2] 2/3 Ok Quit Generated C4.5 Tree Weka Implementation Accuracy: 87.1% The tree took 24 milliseconds to build Our Python Implementation Accuracy: 90.0% The tree took 2 minutes and 45 seconds to build Quit
Dataset: **beer.csv** Train/test split: **1/3** Accuracy: **86%** Weka Accuracy: **85.44%**	 Select Train/Test Split Data file: beer.csv Select Train/Test Data Split [✓ 1/3] 1/2 2/3 Ok Quit Generated C4.5 Tree Weka Implementation Accuracy: 85.44% The tree took 36 milliseconds to build Our Python Implementation Accuracy: 86.0% The tree took 2 minutes and 5 seconds to build Quit	Dataset: **beer_training.csv** Train/test split: **1/3** Accuracy: **70%** Weka Accuracy: **75.9%**	 Select Train/Test Split Data file: beer_training.csv Select Train/Test Data Split [✓ 1/3] 1/2 2/3 Ok Quit Generated C4.5 Tree Weka Implementation Accuracy: 75.9% The tree took 29 milliseconds to build Our Python Implementation Accuracy: 70.0% The tree took 1 minutes and 18 seconds to build Quit

In a Separate file - 10tests.py - We ran the build tree functions for our implementation and the weka implementation 10 times with different split fractions, and stored the results in files. We then plotted these results generated by these runs on graphs, Mapping the split fraction to the accuracy.





Conclusions and Observations

From the Graphs shown above, we can see a definite positive correlation between Accuracy and split fraction. This does seem to level out at a Split Fraction of about 0.5 (or ½) for both the python implementation and the Weka implementation. From this we can gather that, for the beer dataset of 154 attributes, roughly 77 random attributes are enough to generate a reliable accuracy of 80% to 85%.
When the size of the training data falls below 20% of the total file (~46 lines), the created tree will be much less accurate, as the training dataset does not provide a clear enough picture of the entire data set.

We can see that when the algorithm is run on the beer_training.csv dataset from Assignment 1, the trees generated for the ⅔ and ½ data split values are similarly accurate to the trees generated for those same split values from the beer.csv dataset. The tree generated for the ⅓ data split is significantly less accurate with the beer_training data set than it was with the beer data set, and this is likely due to the fact that the beer_training dataset is smaller than the beer dataset (124 lines). Thus, the training dataset would be smaller and less accurate when generated with this smaller dataset.

The Weka implementation performed similarly to the python-from-scratch implementation in all of the tested scenarios, with less than a 6% difference in accuracy for all tests run through the GUI and through the 10tests.py file. For all scenarios however, the Weka implementation was much faster than our python-from-scratch implementation as the Weka implementation was built with a greater focus on efficiency. The python implementation took between 2 and 5 seconds per training data row when calculating this test data. This time is understandable given the complexity of the algorithm, and the number of recursive calls required to make it work, along with the fact that the tree was being built as a multiprocess.

In every scenario run, the python implementation builds a different tree to the tree built by our implementation from scratch, despite building from identical datasets. This can be attributed to the fact that the Weka implements a more optimised c4.5 algorithm called j48 [9], while our code builds a more basic implementation of the C4.5 algorithm. Despite the trees having different structures, they perform similarly and successfully on the testing data set.

Dependencies

Pip Modules - numpy, pandas, graphviz, javabridge, python-weka-wrapper3, PySimpleGUI, weka
To initialize graphviz for use in generating graphs, open command prompt as admin, type ‘dot -c’, and then restart your code env

Work Division between Team Members

We helped each other out, and paired to solve a lot of issues we were facing in the development of this C4.5 implementation. We did each lead the development for different aspects of the code, as shown in the breakdown below:

function	Written by	function	Written by
main()	louise	split_data_styles(data)	Aideen
processTimes(time)	Aideen	split_data_training_testing(data, ratio)	Aideen
createTreeWhileShowingLoadingWindow(data, split)	Aideen	gather_data()	louise
createTree(data, data_split, quit, queue)	louise	entropy(dataset)	Aideen
renderLoadingWindow(quit)	Aideen	information_gain(train_target, subsets)	louise
getInputData()	Aideen	test_tree(root_node, testing_data, split)	Aideen
errorWindow(text)	Aideen	test_data(data, node, test_results)	Aideen
resize_images()	Aideen	test_lr(node, row)	louise
DisplayTreesPopup(python_accuracy, weka_accuracy, p_time, w_time)	Aideen	class Node	louise
build_tree(data, attributes)	louise	if __name__ == '__main__':	louise
print_tree(root_node)	louise	build_weka_tree(split)	louise
addEl(node, g, rootname)	louise	build_weka(split)	louise
find_best_attribute(train_data, attributes)	louise	10tests.py	Aideen
getMajorityClass(data)	louise		
split_into_subsets(column_header, training_data)	louise		

References

[1] Explanation of C4.5 Algorithm which we studied before beginning to construct our code [Accessed 13 November 2020].
https://towardsdatascience.com/what-is-the-c4-5-algorithm-and-how-does-it-work-2b971a9e7db0

[2] More In-depth explanation of Information gain studied while writing the code [Accessed 13 November 2020].
https://machinelearningmastery.com/information-gain-and-mutual-information/

[3] Python Weka Wrapper documentation used in building the reference algorithm [Accessed 15 November 2020].
https://fracpete.github.io/python-weka-wrapper3/index.html

[4] PySimpleGUI documentation used in building the Python User Interface [Accessed 16 November 2020].
https://pysimplegui.readthedocs.io/en/latest/

[5] Python Multiprocessing documentation used when adding the multiprocessing component to animate the loading gif while the tree is built [Accessed 16 November 2020].
https://docs.python.org/2/library/multiprocessing.html

[6] Machine learning with python: Decision Trees in python [Accessed 14 November 2020].
https://www.python-course.eu/Decision_Trees.php

[7] Examples - graphviz 0.15 documentation [Accessed 17 November 2020].
https://graphviz.readthedocs.io/en/stable/examples.html#btree-py

[8] Decision Trees - C4.5 | octavians blog [Accessed 19 November 2020].
https://octaviansima.wordpress.com/2011/03/25/decision-trees-c4-5/

[9] ResearchGate. 2020. *Figure 1. The Typical Decision Tree Model. J48 Are The Improved*.... [online] Available at: <https://www.researchgate.net/figure/The-typical-decision-tree-model-J48-are-the-improved-versions-of-C45-algorithms-or-can_fig1_288807267#:~:text=decision%20tree%20model.-,J48%20are%20the%20improved%20versions%20of%20C4.,J48%20is%20the%20Decision%20tree.> [Accessed 2 December 2020].