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.

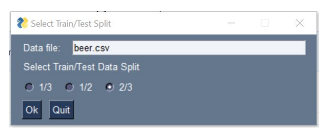
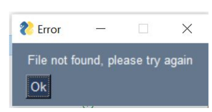
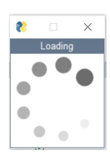
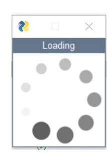
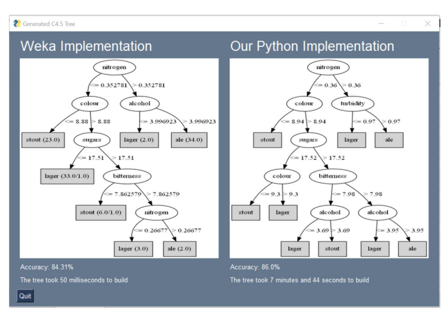
Design Decisions

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

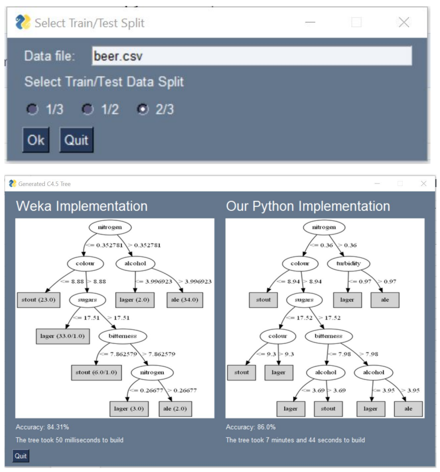
Our implementation of the C4.5 Algorithm runs with a simple Graphical User Interface built using the PySimpleGUI python library.

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	
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	
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.	

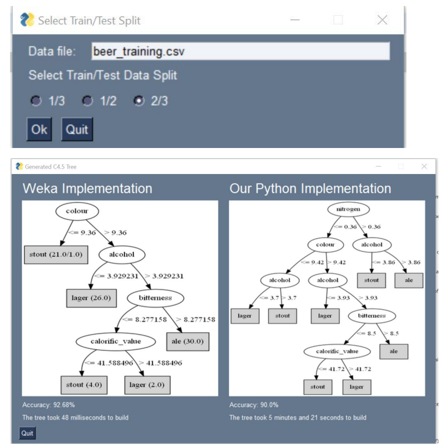
Tests and Results

<table>
<tr><th>Data</th><th>Output</th><th>Data</th><th></th></tr>
<tr>
<td>Dataset:
beer.csv

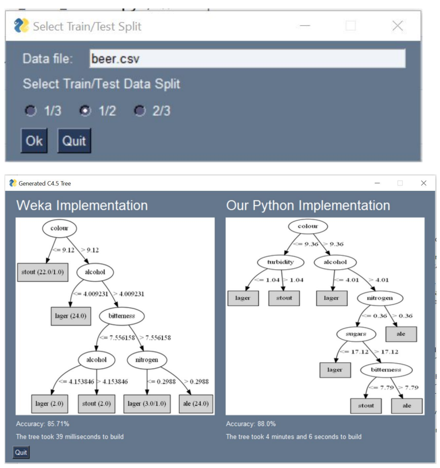
Train/test split: **2/3**

Accuracy: **86%**
Weka Accuracy:
88.24%</td>
<td>
</td>
<td>Dataset:
beer_training.csv

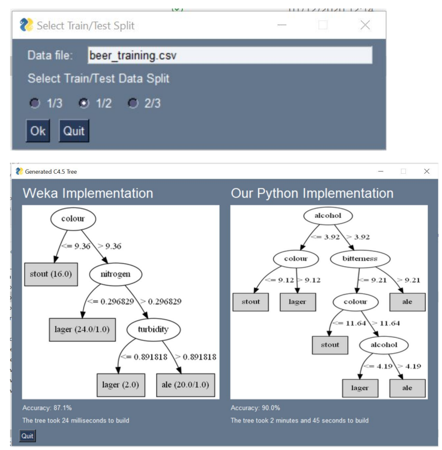
Train/test split: **2/3**

Accuracy: **90%**
Weka Accuracy:
85.37%</td>
<td>
</td>
</tr>
<tr>
<td>Dataset:
beer.csv

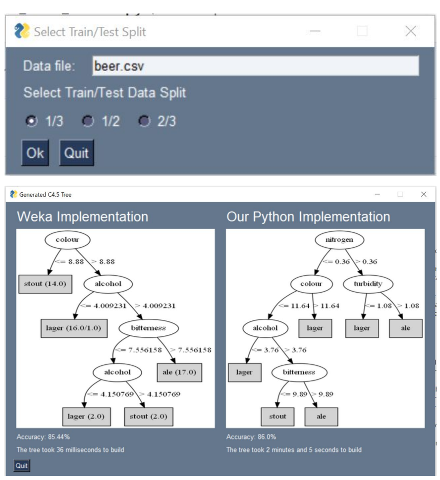
Train/test split: **1/2**

Accuracy: **88%**
Weka Accuracy:
87.01%</td>
<td>
</td>
<td>Dataset:
beer_training.csv

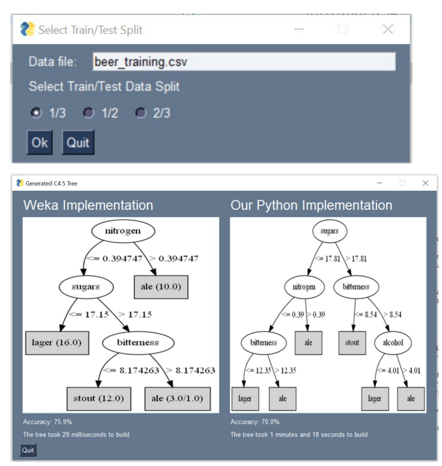
Train/test split: **1/2**

Accuracy: **90%**
Weka Accuracy:
87.1%</td>
<td>
</td>
</tr>
<tr>
<td>Dataset:
beer.csv

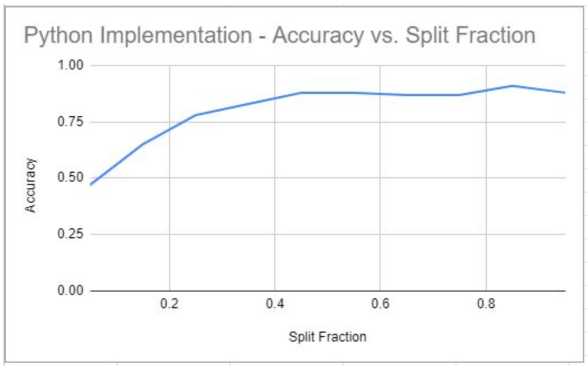
Train/test split: **1/3**

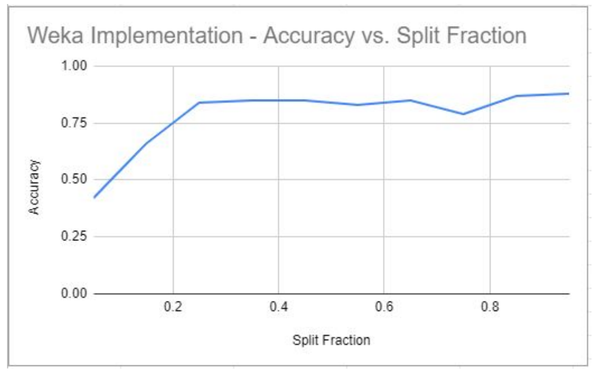
Accuracy: **86%**
Weka Accuracy:
85.44%</td>
<td>
</td>
<td>Dataset:
beer_training.csv

Train/test split: **1/3**

Accuracy: **70%**
Weka Accuracy:
75.9%</td>
<td>
</td>
</tr>
</table>

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].

```python
# implementation_from_scratch.py
# Main file for the implementation of the C4.5 algorithm
import pandas as pd
import numpy as np
import graphviz
from graphviz import Digraph
from math import log2
from copy import deepcopy
from weka_implementation import build_weka_tree
import PySimpleGUI as sg
from PIL import Image, ImageTk, ImageSequence
from multiprocessing import Process, Queue
import multiprocessing as mp
import time
import os.path
from os import path
import PIL.Image
import io
import base64
import math


# louise Kilheeney -16100463
def main():

    # Get the data, the train/test split percentage and the filepath of the data location
    data, split = gather_data()

    # While animate a 'loading' gif to show that the process is running,
    # Build a C4.5 tree from the data.
    # Return the root node of the tree, the dataset to use in testing and the time it took to build the tree
    root_node, testing_data, python_time_to_build = createTreeWhileShowingLoadingWindow(data, split)

    # Draw the tree and store it in png format
    print_tree(root_node)

    # Calculate the accuracy of the built tree using the testing data
    python_accuracy = test_tree(root_node, testing_data, split)

    # Build a tree using the same data in weka.
    # Draw the weka tree and store it in png format
    # Get the accuracy of the weka tree, and the time it took to construct
    weka_accuracy, weka_time_to_build = build_weka_tree(split)

    # Scale and format the times it took to run each implementation
    python_time = processTimes(python_time_to_build)
    weka_time = processTimes(weka_time_to_build)

    # Display both trees side to side, with their accuracies, and the time it took to build them
    DisplayTreesPopup(python_accuracy, weka_accuracy, python_time, weka_time)


# Aideen McLoughlin - 17346123
# Take in the time it took to run a process in seconds
# and return a nicely scaled representation of that value
def processTimes(time):
    if time < 1:
        scaledtime = round(time * 1000)
        timemsg = str(scaledtime) + " milliseconds"
    elif time <= 60:
        timemsg = str(round(time)) + " seconds"
    else:
        minutes = math.floor(time/60)
        seconds = round(time - minutes*60)
```

```python
        timemsg = str(minutes) + " minutes and "+str(seconds)+" seconds"
    return "The tree took " + timemsg + " to build"


# Aideen McLoughlin - 17346123
# Using python multiprocessing, build a tree while showing a 'loading' animation
def createTreeWhileShowingLoadingWindow(data, split):

    # Create an 'Event' to indicate when the tree is built
    quit = mp.Event()
    # Create a 'Queue' to store generated values in
    Q = Queue()

    # Define both processes in the multiprocessing - the tree creation and the loading animation
    p1 = Process(target = createTree, args=(data, split, quit, Q,))
    p2 = Process(target = renderLoadingWindow, args=(quit, ))

    # Store the time before starting to build the tree
    starttime = time.time()

    # Start both processes
    p1.start()
    p2.start()

    # Wait for the quit event to be set, whch will happen once the tree is built
    quit.wait()

    # Store the time once the tree has been built
    endtime = time.time()

    # Get the data stored in the Queue
    queue_data = Q.get()

    # Return the Queue data and the time to build
    return queue_data[0], queue_data[1], endtime-starttime


# louise Kilheeney -16100463
def createTree(data, data_split, quit, queue):

    #call function to split the data into training and test data.
    train_data, test_data = split_data_training_testing(data, (data_split))

    #list of attributes in the data
    attributes = ['calorific_value','nitrogen','turbidity','style','alcohol','sugars','bitterness','colour','degree_of_fermentation']

    #calling function to build tree with the traning data and list of attributes
    root_node = build_tree(train_data, attributes)

    queue.put([root_node, test_data])
    queue.cancel_join_thread()
    quit.set()
    return True


# Aideen McLoughlin - 17346123
def renderLoadingWindow(quit):
    # Declare the PySimpleGUI layout for the popup window
    layout_loading = [[sg.Text("Loading")],[sg.Image(r'loading.gif', key='-IMAGE-')]]

    # Create the popup window
    window = sg.Window("Building C4.5 Tree", layout_loading, element_justification='c', margins=(0,0), element_padding=(0,0), finalize=True)

```

```python
    # Animate the loading gif for the duration of time that 'quit' is not set
    interframe_duration = Image.open(r'loading.gif').info['duration']
    while not quit.is_set():
        event, values = window.read(timeout=interframe_duration)
        if event == sg.WIN_CLOSED:
            exit()
            break

        window.FindElement("-IMAGE-").UpdateAnimation("loading.gif",time_between_frames=interframe_duration)

    # Close the popup window
    window.close()
    return True


# Aideen McLoughlin - 17346123
def getInputData():
    # Declare the PySimpleGUI layout for the popup window
    layout = [
        [sg.Text('Data file: '), sg.InputText("beer.csv", key="file")],
        [sg.Text("Select Train/Test Data Split")],
        [sg.Radio('1/3',"1", key="1/3"),
           sg.Radio('1/2',"1", key="1/2"),
           sg.Radio('2/3',"1", key="2/3", default=True)],
         [sg.Button('Ok'), sg.Button('Quit')]]


    # Create the popup window
    window = sg.Window("Select Train/Test Split", layout)

    # Loop until the window is closed with 'x' 'OK' or 'Quit'
    # Set split to be the selected train/test split value
    # And set the filepath to be the contents of the InputText box (default beer.csv)
    while True:
        event, values = window.read()
        if event == sg.WIN_CLOSED or event == 'Quit':
            break
        if event == 'Ok':
            if values["1/3"] == True:
                split = (1/3)
            elif values["1/2"] == True:
                split = (1/2)
            elif values["2/3"] == True:
                split = (2/3)
            break
    filepath = values["file"]

    # Close the popup window
    window.close()
    return split, filepath


# Aideen McLoughlin - 17346123
def errorWindow(text):
    # Declare the PySimpleGUI layout for the popup window
    layout = [[sg.Text(text)],[sg.Button('Ok')]]

    # Create the popup window
    window = sg.Window("Error", layout)

    # Display a popup window with the text passed as a function param, until the user closes the window
    while True:
        event, values = window.read()
        if event == sg.WIN_CLOSED or event == 'Ok':
            break
```

```
    # Close the popup window
    window.close()


# Aideen McLoughlin - 17346123
# Resixe the tree png images to be equal and fit nicely into the popup window
def resize_images():
    for image in (r'weka-test.gv.png', r'test.gv.png'):
        img = PIL.Image.open(image)
        img = img.resize((400, 400), PIL.Image.ANTIALIAS)
        img.save(image, format="PNG")


# Aideen McLoughlin - 17346123
def DisplayTreesPopup(python_accuracy, weka_accuracy, p_time, w_time):

    # Resize the tree images to be the right size for the popup
    resize_images()

    # Define 2 columns for the layout
    # The right one for the custome python implementation from scratch
    # The left one for the implementation with weka
    weka_column = [
        [sg.Text("Weka Implementation",font=('Helvetica 20'))],
        [sg.Image(r'weka-test.gv.png',key='-IMAGE-')],
        [sg.Text("Accuracy: "+str(weka_accuracy)+"%")],
        [sg.Text(w_time)]
    ]
    python_column = [
        [sg.Text("Our Python Implementation",font=('Helvetica 20'))],
        [sg.Image(r'test.gv.png',key='-IMAGE-')],
        [sg.Text("Accuracy: "+str(python_accuracy*100)+"%")],
        [sg.Text(p_time)]
    ]

    # Declare the PySimpleGUI layout for the popup window with the two columns, and a QUIT button
    layout = [
        [
            sg.Column(weka_column),
            sg.Column(python_column)
        ],
        [sg.Button('Quit')],
    ]

    # Create the popup window
    window = sg.Window("Generated C4.5 Tree", layout)

    # Display the popup window  until the user closes it
    while True:
        event, values = window.read()
        if event == sg.WIN_CLOSED or event == 'Quit': # if user closes window or clicks cancel
            break

    # Close the popup window
    window.close()


# louise Kilheeney - 16100463
def build_tree(data, attributes):
    #1. check the base cases
        #•  All the examples from the training set belong to the same class ( a tree leaf labeled with that class is returned ).
    if data['style'].nunique() == 1:
         return Node(True, data['style'][0], None)
```

```
    #•  The training set is empty ( returns a tree leaf called failure ).
    if len(data) == 0:
        return Node(True, "Fail", None)

    #2. find attribute with highest info gain, retrun best_attribute
        # calling function find-best-attribute which retruns the best attribute, the attribute subsets and the threshold divisor
    best_attribute, attribute_subsets, threshold_divisor  = find_best_attribute(data, attributes)

    #if best attribute is empty
    if best_attribute == "":
        #calling function get majorityclass to return the majority class
        majClass = getMajorityClass(data)
        return Node(True, majClass, None)
    else:
        #3. split the set (data) in subsets arrcording to value of best_attribute
        #attribute_subsets = split_into_subsets(best_attribute, data)
        remainColumns = deepcopy(data)
        remainColumns = data.drop(columns=[best_attribute])

        #4. repeat steps for each subset
        node = Node(False, best_attribute, threshold_divisor)
        for attr_subset in attribute_subsets:
            node.children.append(build_tree(attr_subset, remainColumns))

        return node


# Louise Kilheeney - 16100463
# Generate a png of the tree from the root node
def print_tree(root_node):

    # Create a diagraph in which to store the tree data
    g = Digraph('python_tree_implementation')

    # Add the root node, and all its children recursively
    addEl(root_node, g, 'a')

    # Format the graph as a png, and save it
    g.format = "png"
    g.render('test.gv', view=False)


# Louise Kilheeney - 16100463
# Add a node to the tree
def addEl(node, g, rootname):

    # If the node is not a leaf
    if not node.isLeaf:
        #Create the node
        g.node(name=str(rootname), label=node.label)

        # Create an edge from the node to its left child
        nodename1 = rootname+'b'
        g.edge(rootname, nodename1, label="<= "+str(round(node.divisor,2)))
        # Recursively add the nodes left child
        addEl(node.children[0],g,nodename1)

        # Create an edge from the node to its right child
        nodenamec = rootname + 'c'
        g.edge(rootname, nodenamec, label="> "+str(round(node.divisor,2)))
        # Recursively add the nodes right child
        addEl(node.children[1],g,nodenamec)
    else:
        # If the node is a leaf, add it while styling it as a leaf
        g.node(name=rootname, label=node.label, shape='box', style='filled')
```

```python


# Louise Kilheeney - 16100463
def find_best_attribute(train_data, attributes):
    #  Returns the best attribute from all
    best_information_gain = 0
    best_attribute = ""
    threshold_divisor = ""
    subsets = []
    for attribute in attributes:
        #making sure not to include style
        if attribute != 'style':

            #calling function split_into_subsets
            temp_subsets, temp_divisor = split_into_subsets(attribute, train_data)

            # temp gain is equal to the information gain function for the train_data and the subsets.
            temp_gain = information_gain(train_data, temp_subsets)

            #check for the best attribute
            if temp_gain > best_information_gain:
                best_attribute = attribute
                best_information_gain = temp_gain
                subsets = temp_subsets
                threshold_divisor = temp_divisor
    # return the best attribute , subsets and the threshold divisor
    return best_attribute, subsets, threshold_divisor


# Louise Kilheeney - 16100463
# Function to get the majority class of the data been passed in.
def getMajorityClass(data):
    #find the majority class in the data with the data - style
    grouped = data.groupby(data['style'])
    #return majority class
    return max(grouped.groups)


# Louise Kilheeney - 16100463
def split_into_subsets(column_header, training_data):
    split_values = []
    maxEnt = -1*float("inf")
    best_threshold = ""
    sorted_data = training_data.sort_values(by=[column_header])
    for item in range(0, len(training_data) - 1):
        if type(sorted_data.iloc[item][column_header]) != 'style':
            if sorted_data.iloc[item][column_header] != sorted_data.iloc[item+1][column_header]:
                threshold = (sorted_data.iloc[item][column_header] + sorted_data.iloc[item+1][column_header]) / 2
                smaller_than_threshold = pd.DataFrame()
                bigger_than_threshold = pd.DataFrame()
                for index, row in sorted_data.iterrows():
                    if(row[column_header] > threshold):
                        bigger_than_threshold = bigger_than_threshold.append(row, ignore_index = True)
                    else:
                        smaller_than_threshold = smaller_than_threshold.append(row, ignore_index = True)

                igain = information_gain(training_data, [smaller_than_threshold, bigger_than_threshold])

                if igain >= maxEnt:
                    split_values = [smaller_than_threshold, bigger_than_threshold]
                    best_threshold = threshold
```

```
                    maxEnt = igain
    return split_values, best_threshold


# Aideen McLoughlin - 17346123
# Split the python DataFrame object into 3 dataFrame objects
# One storing all the values with the syle 'ale'
# One storing all the values with the syle 'lager'
# and One storing all the values with the syle 'stout'
def split_data_styles(data):

    # Declare empty subsets array
    subsets = {}

    #Group the data passed to the function by its style
    grouped = data.groupby(data['style'])

    # For each style name, add the values in that group to the subsets array as a new Dataframe object
    for index, beer_style in enumerate(['ale','lager','stout']):
        if beer_style in grouped.groups.keys():
            subsets[index] = grouped.get_group(beer_style)
        else:
            subsets[index] = {}

    # return the subsets array of DataFrame objects
    return subsets


# Aideen McLoughlin - 17346123
# Split the data into training and testing datasets
def split_data_training_testing(data, ratio):

    # Drop the beer_id column as it is not relevant to the beer style
    data = data.drop(columns=['beer_id'])

    # Get a random sample from the data file as the training data
    # This data will be ratio% of the initial dataset
    train = data.sample(frac=ratio,random_state=5)

    # Get the rest of the dataset values as the testing data
    test = data.merge(train, how='left', indicator=True)
    test = test[(test['_merge']=='left_only')].copy()
    test = test.drop(columns='_merge').copy()

    # Save the divided dataset so that it can be used in the weka implementation
    train.to_csv('train_data_generated.csv',index=False,header=True)
    test.to_csv('test_data_generated.csv',index=False,header=True)

    # Create a folder to store the reults in if it does not already exist
    if not path.exists('results'):
        print("Create results folder")
        os.mkdir('results')

    # Return the training and testing data
    return train, test

# Louise Kilheeney - 16100463
# Get the filepath of the data file, and the train/test data split fro user imput in a PySimpleGUI popup
# If a filepath provided is not valid, prompt the user to input a new filepath.
# Repeat until a valid filepath is provided
def gather_data():

    # Get the train/test split and the filepath of the data file
    split, filepath = getInputData()

```

```
    # Create an empty pandas dataframe element
    data = pd.DataFrame()

    # While the dataframe element remains empty
    while data.empty:

        # Check If the filepath is valid
        if path.isfile(filepath):
            # If it is, set the data to be the csv data at that filepath
            data = pd.read_csv(filepath)
        else:
            # If it is not valid, Display an error pop-up and prompt the user to imput the split and filepath again
            errorWindow("File not found, please try again")
            split, filepath = getInputData()

    # Once the dataframe element is filled, return the data, the train/test split percentage and the filepath (For use in the weka implementation)
    return data, split


# Aideen McLoughlin - 17346123
# Calculate the entropy of the passed data set
def entropy(dataset):

    # Initialise entropy to zero value
    entropy = 0

    # Get the ale, lager and stout subset DataFrames from the passed dataset
    subsets = split_data_styles(dataset)

    # For each subset
    for index in range(len(subsets)):

        # Get the percentage of the dataset which is in the subset
        probability = len(subsets[index])/ len(dataset)

        # If the probability is not zero,
        # Subtract plog2(p) from the entropy value where p is probability
        if probability != 0:
            entropy = entropy - (probability)*log2(probability)

    # Return the entropy value
    return entropy


# Louise Kilheeney - 16100463
# function to calculate the information gain
def information_gain(train_target, subsets):
    #getting the entropy value of  the train_target
    entropyTarget = entropy(train_target)
    total = len(train_target)

    Gain = entropyTarget

    #for each subset
    for i in range(0, len(subsets)):
        #length of each subset
        numBeer = len(subsets[i])
        #Gain = Ent(S) - |S beer =ale |/|S|*( Ent( S beer=ale )) -  |S beer=stout |/|S|*( Ent( S beer=stout )) - |S beer=lager |/|S|*( Ent( S beer=lager )) - ....
        firstPart = numBeer/total
        secondPart = entropy(subsets[i])
        Gain -= (firstPart*secondPart)
    # Return the information gain value
    return Gain
```

```python

# Aideen McLoughlin - 17346123
# Test the built tree using the testing data
def test_tree(root_node, testing_data, split):

    # Get the style as the target values, and then drop them from the testing dataset
    test_target = testing_data['style'].values
    testing_data = testing_data.drop(columns=['style'])

    # get the results of the tree predictions for all the testing data values
    test_results = test_data(testing_data, root_node, [])

    # Initialise the number of correct entries to 0
    correct = 0

    accurate = []
    # For each test result, check if it is accurate using the style values we removed from the Dataframe earlier
    # Keep a count of the number of correct predictions
    # If the prediction is wrong, print the incorrect predicton
    for index in range(0, len(test_results)):
        if test_results[index] == test_target[index]:
            correct = correct +1
            accurate.append(1)
        else:
            accurate.append(0)

    # Calculate the accuracy to 2 decimal places, and return it
    accuracy = round(correct/len(test_results),2)

    df = pd.DataFrame()
    df['Actual'] = test_target
    df['Predicted'] = test_results
    df['Accuracy'] = accurate

    filename = "results/python-results-"+str(round(split,2))+"-"+str(round(accuracy,2))+".csv"
    df.to_csv(filename,index=False,header=True)

    return accuracy


# Aideen McLoughlin - 17346123
# Using the root node of the constructed tree, predict the output of all the test data inputs
def test_data(data, node, test_results):

    # For each data value, get the predicted result
    for item in range(0, len(data)):
        test_results.append(test_lr(node, data.iloc[item]))

    # return the set of all predicted results
    return test_results


# Louise Kilheeney - 16100463
# Get the final leaf node destination for a data row
def test_lr(node, row):

    # Decide which child path of a node to proceed into, based on the input value.
    # This function will call itself recursively until it reaches a leaf node
    # That leaf node will be returned to the function which called it
    if node.isLeaf:
        return node.label
    else:
        if row[node.label] <= node.divisor:
            return test_lr(node.children[0], row)
```

```
        else:
            return test_lr(node.children[1], row)


# Louise Kilheeney - 16100463
# Node class
class Node:
    def __init__(self,isLeaf, label, divisor):
        self.label = label
        self.isLeaf = isLeaf
        self.divisor = divisor
        self.children = []


# Louise Kilheeney - 16100463
# Set the main function to run when the file is run
if __name__ == '__main__':
    main()
```

```python
# weka_implementation.py

from weka.classifiers import Classifier
from weka.core.converters import Loader
import weka.core.jvm as jvm
from weka.core.dataset import create_instances_from_lists
from weka.filters import Filter
from weka.core.classes import Random
from weka.classifiers import Classifier, Evaluation, PredictionOutput
import weka.plot.graph as graph
import graphviz
import time
import pandas as pd
import numpy as np
from graphviz import Digraph

# Louise Kilheeney - 16100463
# Taking in the data file location, and the train/test split proportion
# Build a weka C4.5 implementation using the Python Weka Wrapper API
def build_weka_tree(split):
    jvm.start()

    accuracy, time_to_build = build_weka(split)

    jvm.stop()
    return accuracy, time_to_build

# Louise Kilheeney - 16100463
def build_weka(split):
    # Load the data file
    loader = Loader(classname="weka.core.converters.CSVLoader")
    train = loader.load_file('train_data_generated.csv')
    test = loader.load_file('test_data_generated.csv')

    # Store the target values for the test data
    # so that the accuracy of the formula can be checked
    test_target = pd.read_csv('test_data_generated.csv')['style'].values
    # Get the dataset used to train the model,
    # so that we can identify what the key values for the class are.
    # As data is split randomly, we cannot assume it is in [ale, lager,stout] order
    train_classes = pd.read_csv('train_data_generated.csv')['style'].values

    # Set the class to be column 3 - the style column
    train.class_index = 3

    # Set the class to be column 3 - the style column
    test.class_index = 3

    # Store the time before starting to build the tree
    starttime = time.time()

    # initialise the time_to_run and accuracy to 0
    time_to_run = 0
    accuracy = 0

    # Build and Train the weka tree
    cls = Classifier(classname="weka.classifiers.trees.J48")

    # Check that the data is valid
    # If so, Build and Train the weka tree
    if len(list(np.unique(train_classes))) != 1:
        cls.build_classifier(train)
        graph = cls.graph
        # Store the time once the tree has been built
        endtime = time.time()

        # Create a list to store the predicted values in
```

```
        pred = []
        accurate = []
        # Get the class labels in the order that they were allocated when training the model
        classes = pd.Series(train_classes).drop_duplicates().tolist()

        correct = 0
        total = 0
        # loop through test dataset, incrementing total every time
        # and incrementing count if the predicted value was correct
        for index, inst in enumerate(test):
            total = total +1
            predicted = classes[int(cls.classify_instance(inst))]
            pred.append(predicted)
            act = test_target[index]
            if predicted == test_target[index]:
                correct = correct+1
                accurate.append(1)
            else:
                accurate.append(0)

        # Get the accuracy of the weka implementation
        accuracy = (correct/total)

        # store the results in a csv file - the predicted class and the actual class
        df = pd.DataFrame()
        df['Actual'] =  test_target
        df['Predicted'] = pred
        df['Accuracy'] = accurate
        filename = "results/weka-results-"+str(round(split,2))+"-"+str(round(accuracy,2))+".csv"
        df.to_csv(filename,index=False,header=True)

        time_to_run = endtime-starttime
    # If the data is invalid, create a node to indicate failure
    elif len(train) == 0:
        graph = Digraph('python_tree_implementation')
        graph.node(name='A', label="Fail", shape='box', style='filled')
    else:
        graph = Digraph('python_tree_implementation')
        graph.node(name='A', label=train_classes[0], shape='box', style='filled')

    # Render a png image of the weka tree to display in the PySimpleGUI popup
    g = graphviz.Source(graph)
    g.format = "png"
    g.render('weka-test.gv', view=False)
    return round(accuracy*100, 2), time_to_run
```

```python
# 10tests.py

# Aideen McLoughlin - 17346123

import pandas as pd
import numpy as np
from implementation_from_scratch import *
from weka_implementation import *
import weka.core.jvm as jvm
# Define a range of train/test split proportions
proportions = [0.05, 0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95]

jvm.start()

# For each proportion
for split in proportions:
    # Read in the data
    data = pd.read_csv('beer.csv')

    # declare Queue for Tree output storage
    Q = Queue()
    # Create tree
    createTree(data, split, mp.Event(), Q)

    # Get the tree output
    queue_data = Q.get()
    root_node = queue_data[0]
    test_data = queue_data[1]

    # Get the accuracy of the tree and test it
    python_accuracy = test_tree(root_node, test_data, split)

    # Build and test the weka tree
    weka_accuracy, weka_time_to_build = build_weka(split)

jvm.stop()
```