# Machine Learning Assignment 2

**Conor Keaney (4BCT)**

For this assignment I have chosen to use the CART algorithm to implement and evaluate.

## CART

CART (Classification and Regression Tree) algorithm is a Binary decision tree, in that each node can have 0-2 child nodes. Each node represents a value and also a split point on that variable in order to guide values down the tree to the leaf nodes. The leaf nodes contain an output which is used to make a prediction (e.g a target label for classification).

The binary decision tree is created by recursively partitioning the input data in the following steps.

1) For all attributes that have yet to be used in the tree, calculate their **Gini impurity** and **information gain** values using the training data.
2) Select the attribute with the highest information gain.
3) Make a tree node containing that attribute.

This node then will partition the data and this algorithm is applied recursively to each partition.

I have chosen to use **Information Gain** as our metric for this project due to my familiarity with it and it is the primary metric used alongside CART. In order to obtain the Information Gain, normally entropy is used, which is the measure of uncertainty , however as is commonplace when using CART, we will be using the **Gini Index**.

The **Gini Index/Impurity** is calculated by subtracting the sum of the squared probabilities of each class from one.  This calculates the amount of uncertainty.

$$\text{Gini} = 1 - \sum_{i=1}^{c} (p_i)^2$$

Fig.1 Gini Index Formula

**Information Gain** lets us quantify how much a query reduces that uncertainty given by the Gini Index. It is calculated by taking the impurity of the starting node from the impurity of the two child nodes.

$$\text{Gain} = \text{CurrentGini} - p*\text{Gini(Left)} - (1-p)*\text{Gini(Right)}$$

Where p is equal to - $\frac{\textit{Length of Left branch}}{\textit{Length of both left and right branches}}$

Fig.2 Information Gain

## Design decisions

A brief overview of the operation of my program is as follows

### Preprocessing

- I used *ctypes* in order to print out a message box to ask the user to add the testing/training files needed for the primary algorithm implementation and the reference implementation.
- *Tkinter* opens the file dialog and the user selects the file(s) required.
- *Ntpath* is used in order to retrieve the name of the file from the filepath, which is then assigned as the filename
- I use my ***load_data*** method in order to load in the files data into the dataset and also to return the number of entries and to assign and return the list of attributes (i.e. *calorific_value, nitrogen, turbidity, style, alcohol, sugars, bitterness, beer_id, colour, degree_of_fermentation* )
- The data is then shuffled and divided into $\frac{2}{3}s$ training data and $\frac{1}{3}$ testing data using the ***split_shuffle*** method.

### Testing

- Rows of data are then inputted into the ***rec_tree*** method which is used to recursively go through the tree in order to build the tree by using the ***split*** function.
- The ***split*** function is used in order to find the best split for data among all attributes by iterating over all attributes and then calculating the information gain for each in the ***gain*** function.
- The Information gain (***gain***) is calculated by subtracting the uncertainty starting node with the Gini Index value of two child nodes.
- The Gini impurity value is calculated by implementing the algorithm discussed in Figure 1.
- The ***fork*** and ***compare*** functions are used in order to test if the current row passed in by split function is greater than or less than the test value which is used to fork the data, directing down the true branch, or the false branch.
- The ***confidence*** method is used in order to determine the percentage confidence of the decisions.
- Once the back-end of the tree has been built by these methods, we can now print the tree with the ***build_tree*** and ***print_leaf*** methods. The tree is drawn within the console with indentation and arrows in order to mimic a tree structure. At each step it will ask a question and take a step in (e.g.Is degree_of_fermentation > 9.48 ?) with another question for the true branch and a different question for the false branch, which is determined in these earlier functions as discussed above based on their Information gain value.
- In my main function I run a while loop 10 times in order to have 10 different interactions all printed out and from these the average accuracy is taken for both my implementation of the algorithm along with the reference implementation. In this case I used scikit-learn's **DecisionTreeClassifier.**
- The printed output of this was written to a file. This included all 10 trees and percentage accuracies along with the overall average percentage accuracy of the implementation and the reference algorithm.

**Results**

The outputted results of 100 iterations (Figure 4) of my program and a sample tree (Figure 3) are as follows (10 runs, with 10 iterations per run):

```
Is nitrogen > 0.360463403 ?
--> True:
  Is sugars > 4.015384615 ?
  --> True:
    Predict {'ale': 31}
  --> False:
    Is calorific_value > 44.59734513 ?
    --> True:
      Predict {'stout': 1}
    --> False:
      Predict {'lager': 1}
--> False:
  Is turbidity > 1.800909091 ?
  --> True:
    Is beer_id > 10.31873684 ?
    --> True:
      Is calorific_value > 45.70353982 ?
      --> True:
        Predict {'lager': 1}
      --> False:
        Predict {'ale': 1}
    --> False:
      Predict {'stout': 24}
  --> False:
    Is sugars > 3.941538462 ?
    --> True:
      Is degree_of_fermentation > 11.52 ?
      --> True:
        Is bitterness > 17.05 ?
        --> True:
          Predict {'ale': 4}
        --> False:
          Predict {'lager': 4}
      --> False:
        Predict {'stout': 6}
    --> False:
      Is bitterness > 19.72 ?
      --> True:
        Predict {'stout': 1}
      --> False:
        Predict {'lager': 29}
```

Fig.3 Sample tree output

| Run # (set of 10) | % Accuracy of my implementation | % Accuracy of reference algorithm |
|---|---|---|
| 1 | 83.333 | 80.333 |
| 2 | 84.313 | 85 |
| 3 | 82.157 | 82.667 |
| 4 | 85.49 | 76.667 |
| 5 | 83.333 | 80 |
| 6 | 82.941 | 75.333 |
| 7 | 87.255 | 72.333 |
| 8 | 83.333 | 80 |
| 9 | 84.117 | 71.667 |
| 10 | 86.078 | 74 |

Fig.4 Accuracy results for 100 iterations

| Average % Accuracy of my implementation | Average % Accuracy of reference algorithm |
|---|---|
| 84.235% | 77.8% |

These details will be outputted to a file called output.txt.

Results can be obtained manually also by running the program and selecting the training / testing files you require.

I used my implementation of DecisionTreeClassifier from Assignment 1 for the reference algorithm. In order to keep correct formatting of the original reference implementation I split data from the beer file into training and testing data files in order to be read in for the x,y testing and training data sets.

**Conclusion**

This is a simple implementation of the C4.5 algorithm. While I found it very difficult at the beginning however it began to make more sense over time and now I think it was a great learning experience and hopefully one I will carry through to my Final Year Project and in the future workplace.

We can see from the results that the average accuracy of my implementation of the algorithm is higher than my reference implementation of the algorithm.

```python
import csv
import random
import sys
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
from tkinter import filedialog
import tkinter as tk
import ntpath
import ctypes


# Used to load in the file data into the dataset
# Also returns the number of entries in the file
def load_data(file):
    d = []
    att = []
    with open(file, 'r') as f:
        r = csv.reader(f, delimiter='\t')
        i = 0
        for row in r:
            if i == 0:
                att = row
                i += 1
            else:
                d.append(row)
                i += 1
    return d, i, att


# Used to shuffle and split the data into 2/3s training data and 1/3 testing data

def split_shuffle(ds, parts):
    random.shuffle(ds)
    p = (len(ds)) // parts
    test = ds[:p]
    train = ds[p:]
    return test, train


# Used to returns all values for a set column

def get_column(rows, col):
    cols = []

    for row in rows:
        cols.append(row[col])
    return cols


# Used to count the number of each label/feature (beer_style) in the rows passed to the function

def y_count(r):
    y_num = {}
    for row in r:
        y = row[label_col]
        if y not in y_num:
            y_num[y] = 0
        y_num[y] += 1
    return y_num


# Used to compare and test if the current row is greater than or equal to the test value
# in order to split up the data

def compare(r, test_c, test_val):
```

```python
    if r[test_c].isdigit():
        return r[test_c] == test_val

    elif float(r[test_c]) >= float(test_val):
        return True

    else:
        return False


# Splits the data into two lists for the true/false results of the compare test
def fork(r, c, test_val):
    true = []
    false = []

    for row in r:

        if compare(row, c, test_val):
            true.append(row)
        else:
            false.append(row)

    return true, false


# Used to calculate the Gini Index/Impurity of the rows inputted (of beer style)

def gini_index(r):
    stylesNum = y_count(r)
    impurity = 1

    for style in stylesNum:
        style_prob = stylesNum[style] / float(len(r))
        impurity -= style_prob ** 2
    return impurity


# Used to calculate the Information gain, incorporates the gini index (impurity)

def gain(left, right, impurity):
    p = float(len(left)) / (len(left) + len(right))
    ig = impurity - p * gini_index(left) - (1 - p) * gini_index(right)
    return ig


# Used to find the best split for data among all attributes

def split(r):
    max_ig = 0
    max_att = 0
    max_att_val = 0

    # calculates gini for the rows provided
    curr_gini = gini_index(r)
    no_att = len(r[0])

    # Goes through the different attributes

    for c in range(no_att):

        # Skip the label column (beer style)

        if c == label_col:
            continue
        column_vals = get_column(r, c)

        i = 0
```

```
        while i < len(column_vals):
            # value to compare
            att_val = r[i][c]

            # Use the attribute value to fork the data to true and false streams
            true, false = fork(r, c, att_val)

            # Calculate the information gain
            ig = gain(true, false, curr_gini)

            # If this gain is the highest found then mark this as the best choice
            if ig > max_ig:
                max_ig = ig
                max_att = c
                max_att_val = r[i][c]
            i += 1

    return max_ig, max_att, max_att_val


# Used to recursively go through the tree in order to find the optimal attribute to split the tree with

def rec_tree(r):
    ig, att, curr_att_val = split(r)

    if ig == 0:
        return Leaf(r)

    true_rows, false_rows = fork(r, att, curr_att_val)

    true_branch = rec_tree(true_rows)
    false_branch = rec_tree(false_rows)

    return Node(att, curr_att_val, true_branch, false_branch)


# Defines the classifications of the leaf

class Leaf:
    def __init__(self, rows):
        self.predictions = y_count(rows)


# Defines a split node - contains the primary attribute its value and the two child branches

class Node:
    def __init__(self, att, att_value, true_branch, false_branch):
        self.att = att
        self.att_value = att_value
        self.true_branch = true_branch
        self.false_branch = false_branch


# Confidence is used in order to determine what is each value

def confidence(r, node):
    if isinstance(node, Leaf):
        return node.predictions

    c = node.att
    att_value = node.att_value

    if compare(r, c, att_value):
        return confidence(r, node.true_branch)
    else:
```

```
            return confidence(r, node.false_branch)


# Prints and formats the tree based on the branches and questions

def build_tree(node, spacing=""):
    # If you've reached the terminal state then predict
    if isinstance(node, Leaf):
        print(spacing + "Predict", node.predictions)
        return

    print(spacing + "Is " + attributes[node.att] + " > " + str(node.att_value) + " ?")

    print(spacing + '--> True:')
    build_tree(node.true_branch, spacing + "  ")

    print(spacing + '--> False:')
    build_tree(node.false_branch, spacing + "  ")


# Prints out the leaf (the beer style)

def print_leaf(counts):
    total = sum(counts.values())
    probs = {}
    for lbl in counts.keys():
        probs[lbl] = str(int(counts[lbl] / total * 100)) + "%"
    return probs

# Ntpath is used in order to retrieve the name of the file from the file path

def path_name(path):
    head, tail = ntpath.split(path)
    return tail or ntpath.basename(head)


if __name__ == "__main__":
    #TKinter is used in order to open file dialog to get the training and testing data
    root = tk.Tk()
    root.withdraw()
    #ctypes is used in order to print out a message box to tell the user which files are being asked of them
    ctypes.windll.user32.MessageBoxW(0, "Select your training + testing data", "File Selection", 0)

    file_path = filedialog.askopenfilename()
    print(file_path)
    filename = path_name(file_path)
    filename="beer.txt"

    #Label col in this case is beer style and is adjustable to whichever attritbute you choose
    label_col = 3
    avg_acc = 0
    avg_ref_acc = 0
    i = 0
    data, classes, attributes = load_data(filename)

    # -------------------------------------------------------------------#
    # This is for the reference implementation of the decision tree classifier


    featuredCols = ['calorific_value', 'nitrogen', 'turbidity', 'alcohol', 'sugars', 'bitterness', 'beer_id',
                    'colour', 'degree_of_fermentation']
    ref_attributes = ['calorific_value', 'nitrogen', 'turbidity', 'beer_style', 'alcohol', 'sugars', 'bitterness',
```

```
                      'beer_id', 'colour', 'degree_of_fermentation']

    ctypes.windll.user32.MessageBoxW(0, "Select your reference algorithm training data", "File Selection", 0)
    ref_train_path = filedialog.askopenfilename()
    train_path_filename = path_name(ref_train_path)

    ctypes.windll.user32.MessageBoxW(0, "Select your reference algorithm testing data", "File Selection", 0)
    ref_test_path = filedialog.askopenfilename()
    test_path_filename = path_name(ref_test_path)

    trainingData = pd.read_csv("training.txt", sep='\t', names=ref_attributes)
    testData = pd.read_csv("test.txt", sep='\t', names=ref_attributes)
    sys.stdout = open('output.txt', 'wt')

    # ------------------------------------------------------------------#
    # Main random divisions of the algorithm. Each time the testing and training data is
    # shuffled and split randomly

    while i < 10:
        testing, training = split_shuffle(data, 3)
        tree = rec_tree(training)
        build_tree(tree)

        correct = 0
        incorrect = 0
        for r in testing:
            print("Actual: %s. Predicted: %s" % (r[label_col], print_leaf(confidence(r, tree))))
            for key, value in confidence(r, tree).items():
                if r[label_col] == key:
                    correct += 1
                else:
                    incorrect += 1
        print('Percentage Correctly Classified')
        print(correct / (correct + incorrect) * 100)
        print('Percentage Incorrectly Classified')
        print(incorrect / (correct + incorrect) * 100)

        i += 1
        avg_acc += correct / (correct + incorrect)

        # ------------------------------------------------------------------#
        # REFERENCE IMPLEMENTATION

        x_train = trainingData[featuredCols]
        y_train = trainingData.beer_style
        x_test = testData[featuredCols]
        y_test = testData.beer_style

        dtc = DecisionTreeClassifier()
        dtc = dtc.fit(x_train, y_train)
        y_predict = dtc.predict(x_test)

        avg_ref_acc += metrics.accuracy_score(y_test, y_predict)
        print('Reference Algorithms Percentage Accuracy')
        print(metrics.accuracy_score(y_test, y_predict)*100)
        # ------------------------------------------------------------------#

    print("\nThe Average Accuracy across 10 iterations: ")
    acc10 = avg_acc / 10 * 100
    print(acc10)

    refAcc = ((avg_ref_acc / 10) * 100)
    print("\nThe Average Accuracy for the reference decision tree classifier across 10 iterations: ")
```

```
    print(refAcc)

```