CT475

Machine Learning & Data Mining

Assignment 3

Name: Taidgh Murray

Student I.D: 15315901

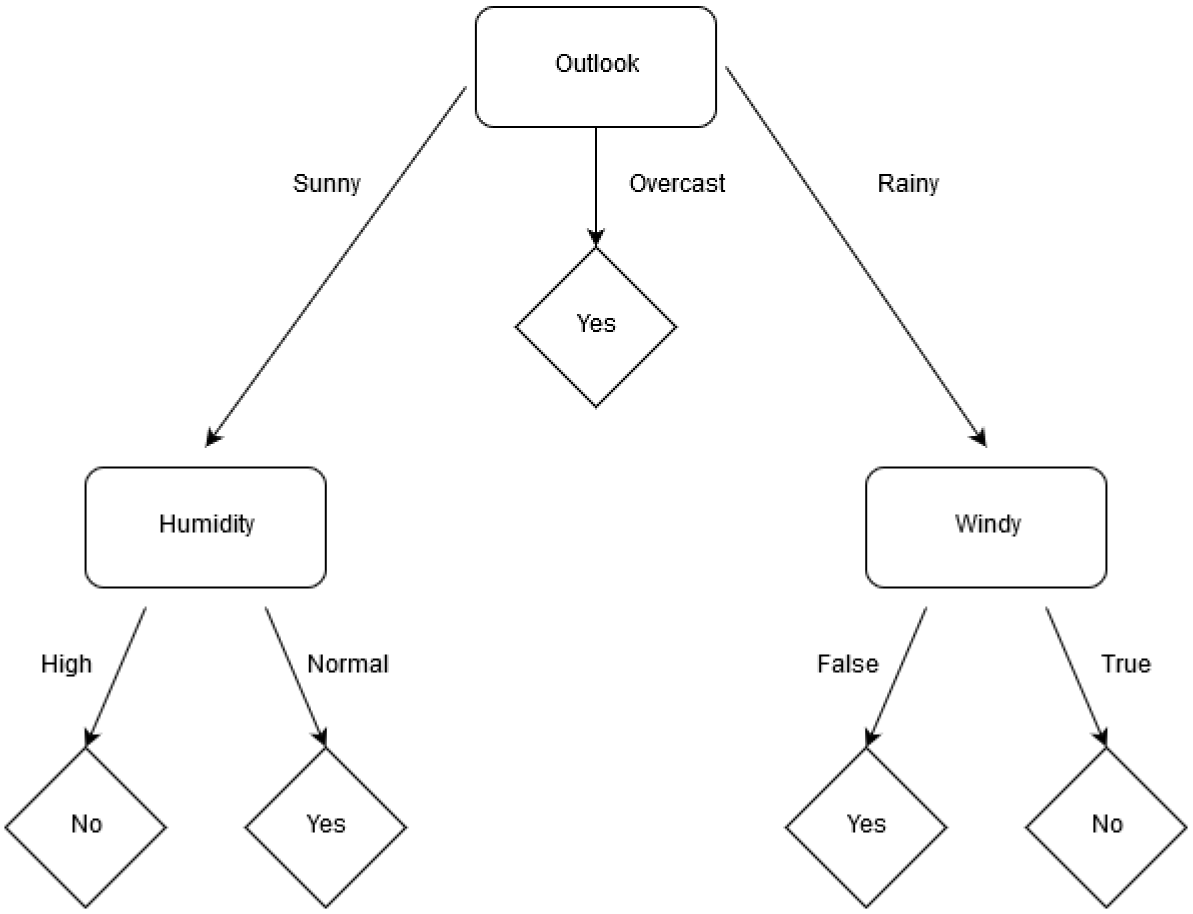
Discipline: College of Science

Course: 4BS2 Undenominated Science (Computing)

In this assignment, we were asked to design, implement, and evaluate a machine learning algorithm, from scratch. The algorithm was to be chosen at our discretion, ensuring to ignore k-Nearest Neighbours, Naïve Bayes, or trivial algorithms such as ZeroR or 1R. It was possible to also create an algorithm from scratch. We were encouraged to work in pairs, but having an introverted, asocial personality, I decided to forego a partner for this assignment, and try to tackle the assignment alone.

As with the previous assignment, I decided to use Python as my language of choice. My experience with it trumps all other programming languages, and the wide variety of libraries available made much of the extraneous programming tasks (mainly reading the ‘owls.csv’ file (or any .csv file the user might want to evaluate the algorithm on), and the random seed generation). Python is simple to understand, and quite portable, which allowed me to easily work on my assignment wherever I could get access to the internet. It also means that reading the code for the algorithm I’ve chosen is much easier than a language like Java, R or MATLAB.

I decided to implement the Classification and Regression Tree algorithm, or CART algorithm. The CART Algorithm can also be called a decision tree algorithm. In its simplest form, the algorithm can be represented with a decision tree. There is an input variable for a node, and a split in the tree based on whether the binary choice is true or false. Below is a simple recreation of the decision tree found in the lecture notes. It is a decision tree for deciding if the weather is good enough to play tennis.



We can see here that if the outlook data is set to ‘Overcast’, then there is a high chance that tennis will be played, so it has a terminal node early. If the weather is sunny or rainy, further queries must be made from the given dataset, to try and give the result more accuracy.

Because the CART algorithm can be represented as a binary tree, it is a very straightforward process to implement it in any programming language of choice. Something along the lines of C would have been ideal for implementation, as its low-level programming and control over the stack and heap would make popping and pushing nodes in a binary tree representation easy, but my proficiency with C is low.

In my implementation, the most important aspect was trying to find the proper splitting point for the data to create a proper binary tree. The best way I found to implement a choosing method for the data was the Gini Impurity. The Gini Impurity is a measure of how likely a given data point in the set will be incorrectly labelled. A formal definition to find the Gini Impurity is as follows:

$$I_g(p) = 1 - \sum_{i=1}^{J} p_i^2$$

Where J is the number of classes in the data. You can see my python implementation of this algorithm in the

```
splitQuality(groups, classes):
```

function. It's interesting to note that the above function had the possibility of dividing by 0. This had given me some issue during my implementation of the algorithm. I had to add an if clause in the code to avoid this happening.

I also needed to implement a stopping criterion for the binary tree. There needs to be a 'cut off' point when building a binary tree, otherwise it will continue indefinitely. The simplest way to implement this would be to code in a stopping point. Adding in a maximum depth and minimum size for the tree. It's a simple process to design the algorithm to implement those variables. I decided to allow the user to customise those variables if they wish, by prompting them for input.

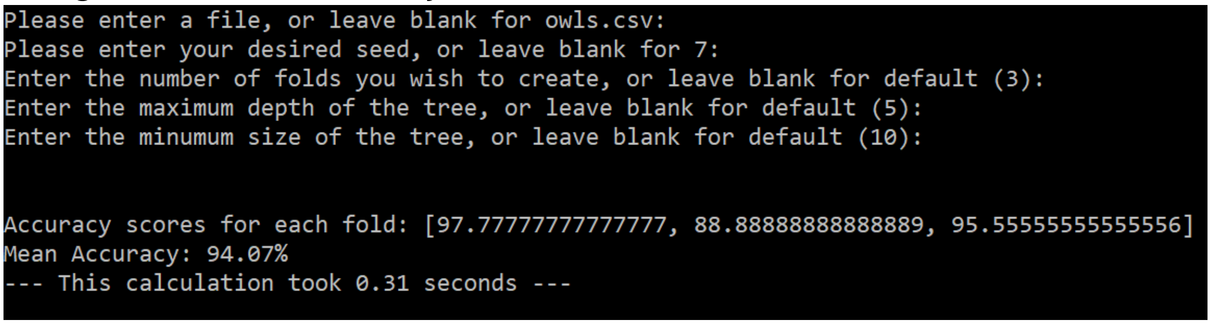
Finally, the last important main task in implementing the CART algorithm is making predictions. Up until now, I've managed to read the data from csv files, split the data and evaluate the splits, and create the binary tree to store the data. With the addition of a terminal node implemented in the algorithm, I simply added a recursive function

```
prediction(node, row):
```

That runs through the tree, checking if the given node is terminal or not.

With the algorithm coded, I decided to add some rudimentary inputs for the user, so that if they wish to test their own files or change the random seed to try and get a more accurate score, they can do so easily. I designed the algorithm to allow this easily, very little (if any) values are hard programmed.

To test the algorithm, I ran the provided 'owls.csv' dataset. Below you can find the results to the algorithm running. The accuracy score for each fold fluctuates from run to run slightly, but on average, each fold stays above 90%, as seen by the mean accuracy calculation. Only in rare cases does a fold slip below 90%, as I managed to showcase in my screenshot.

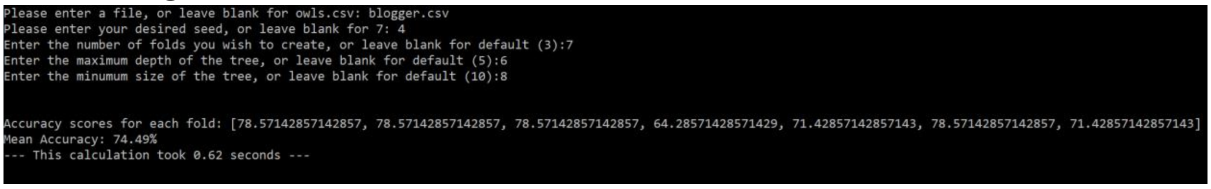


```
Please enter a file, or leave blank for owls.csv:
Please enter your desired seed, or leave blank for 7:
Enter the number of folds you wish to create, or leave blank for default (3):
Enter the maximum depth of the tree, or leave blank for default (5):
Enter the minumum size of the tree, or leave blank for default (10):


Accuracy scores for each fold: [97.77777777777777, 88.88888888888889, 95.55555555555556]
Mean Accuracy: 94.07%
--- This calculation took 0.31 seconds ---
```

I wish to test the algorithm on some other datasets, to make sure that it could be used elsewhere, and was not just an algorithm specifically designed to solve the issue assigned. After all, an important aspect in writing code is making sure it can be as robust as possible.

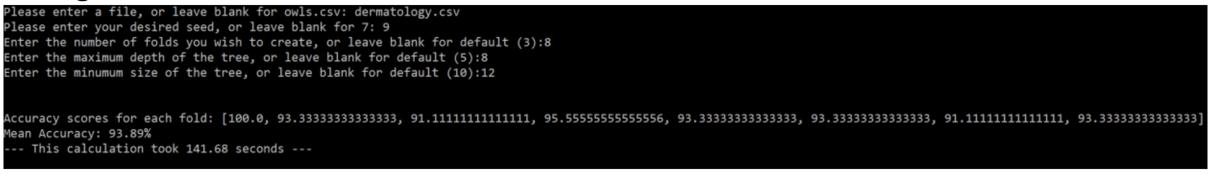
I downloaded a dataset from the machine learning archives of the University of California, Irvine (UCI). I decided on a dataset more text than numbers. I decided on this classification of posts from a blogger[1]. You can find the algorithm's results below, but it seemed to handle the task quite well, although the results are not as reassuring.



```
Please enter a file, or leave blank for owls.csv: blogger.csv
Please enter your desired seed, or leave blank for 7: 4
Enter the number of folds you wish to create, or leave blank for default (3):7
Enter the maximum depth of the tree, or leave blank for default (5):6
Enter the minumum size of the tree, or leave blank for default (10):8


Accuracy scores for each fold: [78.57142857142857, 78.57142857142857, 78.57142857142857, 64.28571428571429, 71.42857142857143, 78.57142857142857, 71.42857142857143]
Mean Accuracy: 74.49%
--- This calculation took 0.62 seconds ---
```

I wanted to see how the algorithm would handle a larger dataset, with more data points, thus meaning more data to handle. I decided on this dataset[2] from dermatologists on various attributes exhibited by their patients. There are 34 separate attributes in this dataset, the size of which was reflected in the speed of the algorithm. Despite taking a considerably longer time, the results of the algorithm are quite accurate, showing an average of 93.89%, which is almost as accurate as the 'owls.csv' dataset this algorithm was designed for.

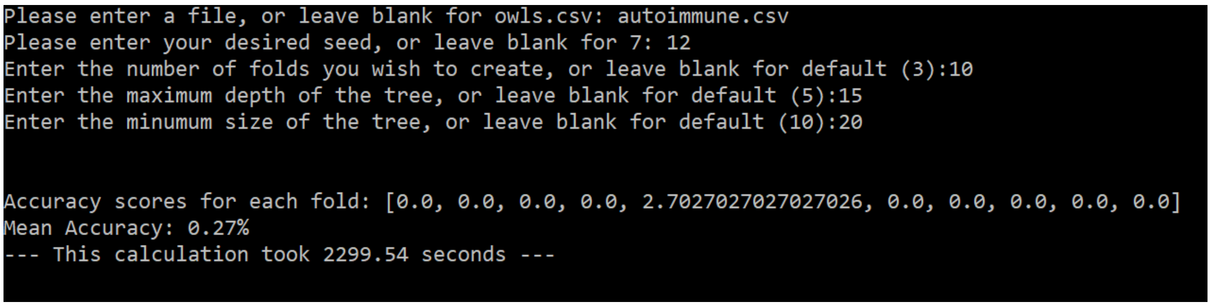


```
Please enter a file, or leave blank for owls.csv: dermatology.csv
Please enter your desired seed, or leave blank for 7: 9
Enter the number of folds you wish to create, or leave blank for default (3):8
Enter the maximum depth of the tree, or leave blank for default (5):8
Enter the minumum size of the tree, or leave blank for default (10):12


Accuracy scores for each fold: [100.0, 93.33333333333333, 91.11111111111111, 95.55555555555556, 93.33333333333333, 93.33333333333333, 91.11111111111111, 93.33333333333333]
Mean Accuracy: 93.89%
--- This calculation took 141.68 seconds ---
```

[1] http://archive.ics.uci.edu/ml/datasets/BLOGGER
[2] http://archive.ics.uci.edu/ml/datasets/Dermatology

I also decided to run my algorithm on the ‘autoimmune.csv’ file I formatted from the previous assignment. The results are interesting, if entirely useless to showcase the algorithm. Even using relatively small values for the inputs, the algorithm takes an enormously long time to attempt a calculation, and the results are not insightful. I felt, however, that the fact the algorithm was so monumentally bad for parsing the ‘autoimmune.csv’ file was in & of itself amusing, so I decided to include it here.



```
Please enter a file, or leave blank for owls.csv: autoimmune.csv
Please enter your desired seed, or leave blank for 7: 12
Enter the number of folds you wish to create, or leave blank for default (3):10
Enter the maximum depth of the tree, or leave blank for default (5):15
Enter the minumum size of the tree, or leave blank for default (10):20


Accuracy scores for each fold: [0.0, 0.0, 0.0, 0.0, 2.7027027027027026, 0.0, 0.0, 0.0, 0.0, 0.0]
Mean Accuracy: 0.27%
--- This calculation took 2299.54 seconds ---
```

It can, however, be tied into my conclusion of the algorithm. As I’ve shown that for relatively small, short datasets, the algorithm does its job very efficiently. Once you try to use much larger datasets, with higher amounts of data points (and thus, decisions the algorithm needs to make), it begins to slow down. This is mainly the case when the number of rows that exist are greater than 10, as is the case with the ‘autoimmune.csv’ file. It would be wiser to use a different, more efficient algorithm in these cases. Provided that the dataset is kept relatively small, then this algorithm works quite nicely.

Below, you can find a copy of the completed algorithm. I’ve also decided to host all files found in a github repo. You can find a link to said repo here.

```
"""
CT475 Assignment 3
Python implementation of CART algorithm.

Name: Taidgh Murray
Student ID: 15315901
Course: 4BS2


Create algorithm (Not Knn or NB)
Write own code
No use of libraries for ML implementation
Must use file input (import csv reader)


Distinguish between 'BarnOwl', 'SnowyOwl' & 'LongEaredOwl'
Divide into 2/3 training, 1/3 testing
Allow for n fold cross validation
Allow users rudimentary input if at all possible
Comment code for ease of reading, and to explain code decisions

```

```python
The main elements of CART algorithm:

    Rules for splitting data at a node based on the value of one variable (Gini Impurity)

    Stopping rules for deciding when a branch is terminal and can be split no more (Values defined by user)

    A prediction for the target variable in each terminal node

"""

# For generating random seed (and other psuedorandom numbers)
import random
# For reading CSV files
import csv
# For printing run time at the end
import time


# For loading csv files
def csvLoader(f):
    f = open(f, "r")
    l = csv.reader(f)
    data = list(l)
    return data


# Randomly (psuedo-randomly) splits 'data' into 'nFolds' amount and creates lists of said splits
def crossValidationSplit(data, nFolds):
    dataSplit = list()
    # dataCopy variable implemented to avoid messy code
    dataCopy = list(data)
    foldSize = int(len(data)/nFolds)
    for j in range(nFolds):
        fold = list()
        while len(fold)<foldSize:
            index = random.randrange(len(dataCopy))
            fold.append(dataCopy.pop(index))
        dataSplit.append(fold)
    return dataSplit

# Accuracy percentage calculation - predicted value vs actual value
def accuracyCalculation(predicted, actual):
    correct = 0
    for k in range(len(actual)):
        if actual[k] == predicted [k]:
            correct+=1
    return ( correct / len(actual) ) * 100

# Evaluates algortihm using the crossValidationSplit function defined above
def algorithmEvaluation(data, algorithm, nFolds, *args):
    folds = crossValidationSplit(data, nFolds)
    score = list()
    for f in folds:
        trainingSet = list(folds)
        trainingSet.remove(f)
        trainingSet = sum(trainingSet, [])
        testingSet = list()
        for row in f:
            rowCopy = list(row)
            testingSet.append(rowCopy)
            rowCopy[-1] = None
        # Calculates predicition score for given algorithm
        pred = algorithm(trainingSet, testingSet, *args)
        act = [row[-1] for row in f]
        accuracy = accuracyCalculation(pred, act)
        score.append(accuracy)
    return score


# Calculate the quality of the data splits - Implementation of Gini Impurity
def splitQuality(groups, classes):
    nInstances = sum([len(g) for g in groups])
    splitQuality = 0
    for g in groups:
        size = len(g)
        # Avoids division by 0
        if size == 0:
            continue
        score = 0
        for c in classes:
            p = [row[-1]for row in g].count(c)/size
            score += p*p
        splitQuality += (1-score)* (size/nInstances)
    return splitQuality

# Splits a dataset based on an attributes
def testingSplit(index, val, data):
    left, right = list(), list()
    for r in data:
        if r[index] < val:
            left.append(r)
        else:
            right.append(r)
    return left, right

# Select the best split for the data, by calculating the splitQuality of the data sets
def getSplit(data):
    c = list(set(row[-1] for row in data))
    splitIndex, splitValue, splitScore, splitGroups = 999, 999, 999, None
    for i in range(len(data[0])-1):
        for row in data:
            # Calls the testingSplit function on the data
            groups = testingSplit(i, row[i], data)
            # Tests the split data for quality
            sQ = splitQuality(groups, c)
            # If the
            if sQ < splitScore:
                splitIndex, splitValue, splitScore, splitGroups = i, row[i], sQ, groups
    return {'index':splitIndex,'value':splitValue,'groups':splitGroups}

# Takes the group of rows assigned to a node and returns the most common value in the group, used to make predictions
def addToTerminal(group):
    outcome = [row[-1] for row in group]
    return max(set(outcome), key = outcome.count)

# Create child nodes for the decision tree
def childNode(node, maxDepth, minSize, depth):
    left, right = node['groups']
    del(node['groups'])

    # If no child nodes exist yet
    if not left or not right:
        node['left'] = node['right'] = addToTerminal(left + right)
        return

    # If the tree can't get any any larger, but the depht returns a larger value
    if depth >= maxDepth:
        node['left'], node['right'] = addToTerminal(left), addToTerminal(right)
        return

    # Left child node
    if len(left) <= minSize:
        # If the left node is smaller than the minimum size, its just added to the tree
        node['left'] = addToTerminal(left)
    else:
        # Otherwise it calls the getSplit function on itself
        node['left'] = getSplit(left)
        # And recursively calls the function on itself, increasing the depth by 1
        childNode(node['left'], maxDepth, minSize, depth+1)

    # Right child node
    if len(right) <= minSize:
        node['right'] = addToTerminal(right)
    else:
        node['right'] = getSplit(right)
        childNode(node['right'], maxDepth, minSize, depth+1)

# Generated initial decision tree
def makeDecisionTree(train, maxDepth, minSize):
    # Starts the tree with the best split of the training data
    root = getSplit(train)
    # Calls childNode function, which will recursively create binary tree from the root node
    childNode(root, maxDepth, minSize, 1)
    return root

# Make prediciton using decision tree
def prediction(node, row):
    # If the index node in the row is smaller than the value node
    if row[node['index']] < node['value']:
        # if the left node is a Python dictionary
        if isinstance(node['left'], dict):
            # Recursively calls the prediction function using the left node and the row
            return prediction(node['left'], row)
        else:
            # Otherwise just returns the left node
            return node['left']

    else:
        if isinstance(node['right'], dict):
            return prediction(node['right'], row)
        else:
            return node['right']


# Calling the CART algorithm
def cart(train, test, maxDepth, minSize):
    # Defines a tree using the makeDecisionTree funcion
    tree = makeDecisionTree(train, maxDepth, minSize)
    # Initialises empty list, called 'predictions' to hold predictions
    predictions = list()
    # Fills prediction list with predictions for each row of info in the test data
    for row in test:
        p = prediction(tree, row)
        predictions.append(p)
    return predictions



```

```
# Allows for some user input, they can chose a different file to be tested, the seed, and change the amount of folds, the maximum depth, and the minimum size of the tree



# Load data
file = (input("Please enter a file, or leave blank for owls.csv: ") or 'owls.csv')
data = csvLoader(file)

# Set Random seed
seed = (input("Please enter your desired seed, or leave blank for 7: ") or 7)
random.seed(seed)


# Evaluate algorithm inputs from user
nFolds = (input("Enter the number of folds you wish to create, or leave blank for default (3):") or 3)
maxDepth = (input("Enter the maximum depth of the tree, or leave blank for default (5):") or 5)
minSize = (input("Enter the minumum size of the tree, or leave blank for default (10):") or 10)

nFolds = int(nFolds)
maxDepth = int(maxDepth)
minSize = int(minSize)

# Record Start Time of prgram
startTime = time.time()

print('\n')
# CART algorithm
scores = algorithmEvaluation(data, cart, nFolds, maxDepth, minSize)

# Formats results and shows the classification accuracy for each fold
print('Accuracy scores for each fold: {}'.format(scores))
# Mean accuracy of accuracy score for CART Algorithm
print('Mean Accuracy: %.2f%%' % (sum(scores)/(len(scores))))

print("--- This calculation took %.2f seconds ---" % (time.time() - startTime))


# Users can then read the completed data before closing the program
input()
```