CT475

Machine Learning & Data Mining

Assignment 3

Name: Taidgh Murray

Student I.D: 15315901

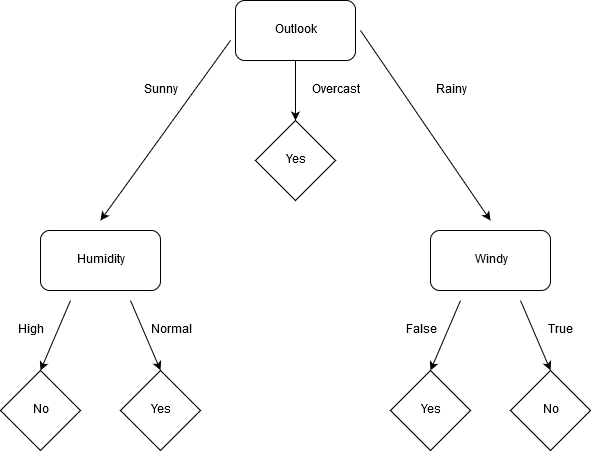
Discipline: College of Science

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:

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.

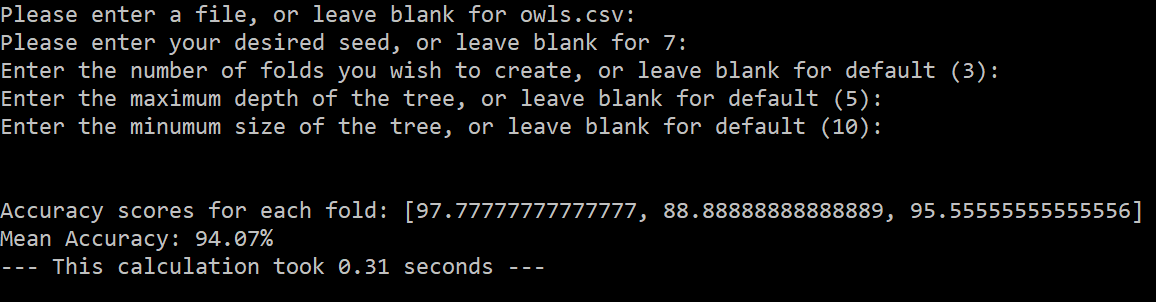
I also needed to implement a stopping criterion for the binary tree. There needs to be a ‘cut off’ point when building a binary tree, otherwise it will continue indefinitely. The simplest way to implement this would be to code in a stopping point. Adding in a maximum depth and minimum size for the tree. 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

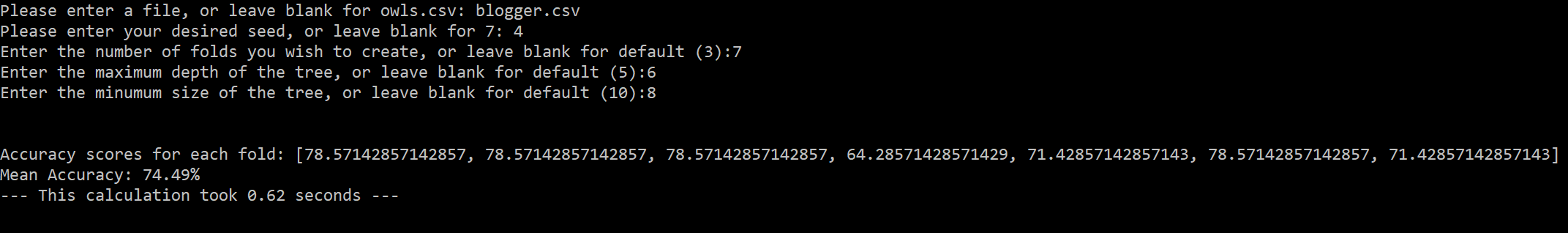
 prediction(node, row):

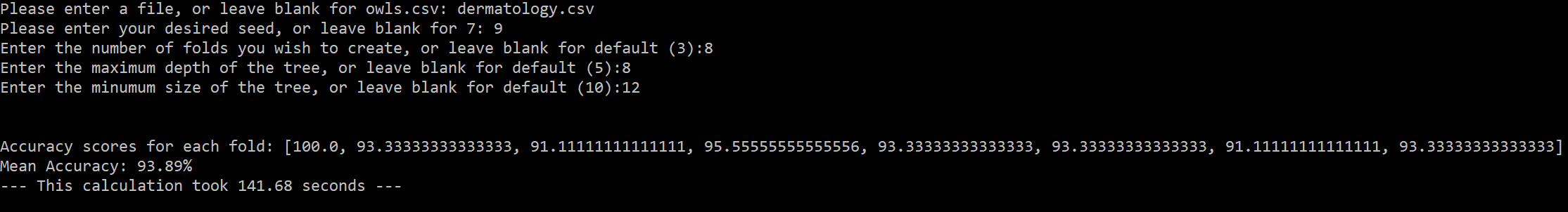
That runs through the tree, checking if the given node is terminal or not.

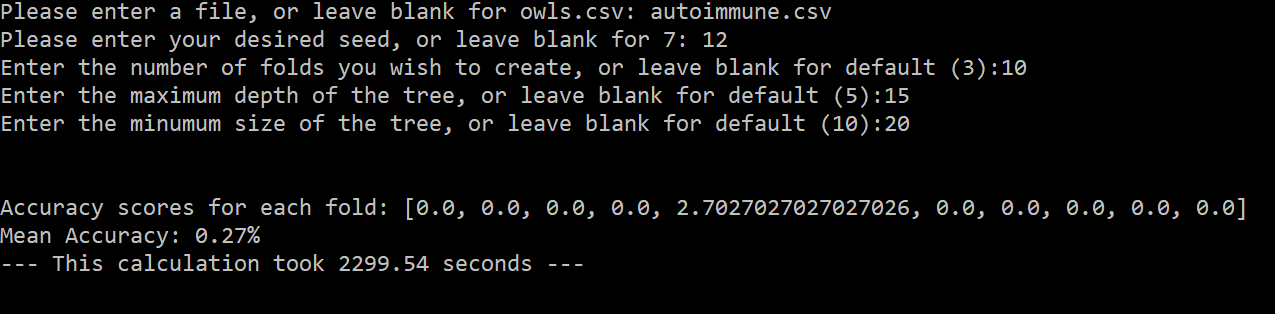
With the algorithm coded, I decided to add some rudimentary inputs for the user, so that if they wish to test their own files or change the random seed to try and get a more accurate score, they can do so easily. I designed the algorithm to allow this easily, very little (if any) values are hard programmed.

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.

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]](#footnote-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.

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]](#footnote-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.

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.

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](https://github.com/tmurray19/CT475-Project).

1. """
2. CT475 Assignment 3
3. Python implementation of CART algorithm.
5. Name: Taidgh Murray
6. Student ID: 15315901
7. Course: 4BS2

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

16. Distinguish between 'BarnOwl', 'SnowyOwl' & 'LongEaredOwl'
17. Divide into 2/3 training, 1/3 testing
18. Allow for n fold cross validation
19. Allow users rudimentary input if at all possible
20. Comment code for ease of reading, and to explain code decisions
22. The main elements of CART algorithm:
24. Rules for splitting data at a node based on the value of one variable (Gini Impurity)
26. Stopping rules for deciding when a branch is terminal and can be split no more (Values defined by user)
28. A prediction for the target variable in each terminal node
30. """
32. # For generating random seed (and other psuedorandom numbers)
33. **import** random
34. # For reading CSV files
35. **import** csv
36. # For printing run time at the end
37. **import** time

40. # For loading csv files
41. **def** csvLoader(f):
42. f = open(f, "r")
43. l = csv.reader(f)
44. data = list(l)
45. **return** data

48. # Randomly (psuedo-randomly) splits 'data' into 'nFolds' amount and creates lists of said splits
49. **def** crossValidationSplit(data, nFolds):
50. dataSplit = list()
51. # dataCopy variable implemented to avoid messy code
52. dataCopy = list(data)
53. foldSize = int(len(data)/nFolds)
54. **for** j **in** range(nFolds):
55. fold = list()
56. **while** len(fold)<foldSize:
57. index = random.randrange(len(dataCopy))
58. fold.append(dataCopy.pop(index))
59. dataSplit.append(fold)
60. **return** dataSplit
62. # Accuracy percentage calculation - predicted value vs actual value
63. **def** accuracyCalculation(predicted, actual):
64. correct = 0
65. **for** k **in** range(len(actual)):
66. **if** actual[k] == predicted [k]:
67. correct+=1
68. **return** ( correct / len(actual) ) \* 100
70. # Evaluates algortihm using the crossValidationSplit function defined above
71. **def** algorithmEvaluation(data, algorithm, nFolds, \*args):
72. folds = crossValidationSplit(data, nFolds)
73. score = list()
74. **for** f **in** folds:
75. trainingSet = list(folds)
76. trainingSet.remove(f)
77. trainingSet = sum(trainingSet, [])
78. testingSet = list()
79. **for** row **in** f:
80. rowCopy = list(row)
81. testingSet.append(rowCopy)
82. rowCopy[-1] = None
83. # Calculates predicition score for given algorithm
84. pred = algorithm(trainingSet, testingSet, \*args)
85. act = [row[-1] **for** row **in** f]
86. accuracy = accuracyCalculation(pred, act)
87. score.append(accuracy)
88. **return** score

91. # Calculate the quality of the data splits - Implementation of Gini Impurity
92. **def** splitQuality(groups, classes):
93. nInstances = sum([len(g) **for** g **in** groups])
94. splitQuality = 0
95. **for** g **in** groups:
96. size = len(g)
97. # Avoids division by 0
98. **if** size == 0:
99. **continue**
100. score = 0
101. **for** c **in** classes:
102. p = [row[-1]**for** row **in** g].count(c)/size
103. score += p\*p
104. splitQuality += (1-score)\* (size/nInstances)
105. **return** splitQuality
107. # Splits a dataset based on an attributes
108. **def** testingSplit(index, val, data):
109. left, right = list(), list()
110. **for** r **in** data:
111. **if** r[index] < val:
112. left.append(r)
113. **else**:
114. right.append(r)
115. **return** left, right
117. # Select the best split for the data, by calculating the splitQuality of the data sets
118. **def** getSplit(data):
119. c = list(set(row[-1] **for** row **in** data))
120. splitIndex, splitValue, splitScore, splitGroups = 999, 999, 999, None
121. **for** i **in** range(len(data[0])-1):
122. **for** row **in** data:
123. # Calls the testingSplit function on the data
124. groups = testingSplit(i, row[i], data)
125. # Tests the split data for quality
126. sQ = splitQuality(groups, c)
127. # If the
128. **if** sQ < splitScore:
129. splitIndex, splitValue, splitScore, splitGroups = i, row[i], sQ, groups
130. **return** {'index':splitIndex,'value':splitValue,'groups':splitGroups}
132. # Takes the group of rows assigned to a node and returns the most common value in the group, used to make predictions
133. **def** addToTerminal(group):
134. outcome = [row[-1] **for** row **in** group]
135. **return** max(set(outcome), key = outcome.count)
137. # Create child nodes for the decision tree
138. **def** childNode(node, maxDepth, minSize, depth):
139. left, right = node['groups']
140. **del**(node['groups'])
142. # If no child nodes exist yet
143. **if** **not** left **or** **not** right:
144. node['left'] = node['right'] = addToTerminal(left + right)
145. **return**
147. # If the tree can't get any any larger, but the depht returns a larger value
148. **if** depth >= maxDepth:
149. node['left'], node['right'] = addToTerminal(left), addToTerminal(right)
150. **return**
152. # Left child node
153. **if** len(left) <= minSize:
154. # If the left node is smaller than the minimum size, its just added to the tree
155. node['left'] = addToTerminal(left)
156. **else**:
157. # Otherwise it calls the getSplit function on itself
158. node['left'] = getSplit(left)
159. # And recursively calls the function on itself, increasing the depth by 1
160. childNode(node['left'], maxDepth, minSize, depth+1)
162. # Right child node
163. **if** len(right) <= minSize:
164. node['right'] = addToTerminal(right)
165. **else**:
166. node['right'] = getSplit(right)
167. childNode(node['right'], maxDepth, minSize, depth+1)
169. # Generated initial decision tree
170. **def** makeDecisionTree(train, maxDepth, minSize):
171. # Starts the tree with the best split of the training data
172. root = getSplit(train)
173. # Calls childNode function, which will recursively create binary tree from the root node
174. childNode(root, maxDepth, minSize, 1)
175. **return** root
177. # Make prediciton using decision tree
178. **def** prediction(node, row):
179. # If the index node in the row is smaller than the value node
180. **if** row[node['index']] < node['value']:
181. # if the left node is a Python dictionary
182. **if** isinstance(node['left'], dict):
183. # Recursively calls the prediction function using the left node and the row
184. **return** prediction(node['left'], row)
185. **else**:
186. # Otherwise just returns the left node
187. **return** node['left']
189. **else**:
190. **if** isinstance(node['right'], dict):
191. **return** prediction(node['right'], row)
192. **else**:
193. **return** node['right']

196. # Calling the CART algorithm
197. **def** cart(train, test, maxDepth, minSize):
198. # Defines a tree using the makeDecisionTree funcion
199. tree = makeDecisionTree(train, maxDepth, minSize)
200. # Initialises empty list, called 'predictions' to hold predictions
201. predictions = list()
202. # Fills prediction list with predictions for each row of info in the test data
203. **for** row **in** test:
204. p = prediction(tree, row)
205. predictions.append(p)
206. **return** predictions


210. # 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


214. # Load data
215. file = (input("Please enter a file, or leave blank for owls.csv: ") **or** 'owls.csv')
216. data = csvLoader(file)
218. # Set Random seed
219. seed = (input("Please enter your desired seed, or leave blank for 7: ") **or** 7)
220. random.seed(seed)

223. # Evaluate algorithm inputs from user
224. nFolds = (input("Enter the number of folds you wish to create, or leave blank for default (3):") **or** 3)
225. maxDepth = (input("Enter the maximum depth of the tree, or leave blank for default (5):") **or** 5)
226. minSize = (input("Enter the minumum size of the tree, or leave blank for default (10):") **or** 10)
228. nFolds = int(nFolds)
229. maxDepth = int(maxDepth)
230. minSize = int(minSize)
232. # Record Start Time of prgram
233. startTime = time.time()
235. **print**('\n')
236. # CART algorithm
237. scores = algorithmEvaluation(data, cart, nFolds, maxDepth, minSize)
239. # Formats results and shows the classification accuracy for each fold
240. **print**('Accuracy scores for each fold: {}'.format(scores))
241. # Mean accuracy of accuracy score for CART Algorithm
242. **print**('Mean Accuracy: %.2f%%' % (sum(scores)/(len(scores))))
244. **print**("--- This calculation took %.2f seconds ---" % (time.time() - startTime))

247. # Users can then read the completed data before closing the program
248. input()

1. http://archive.ics.uci.edu/ml/datasets/BLOGGER [↑](#footnote-ref-1)
2. http://archive.ics.uci.edu/ml/datasets/Dermatology [↑](#footnote-ref-2)