CT475

Machine Learning & Data Mining

Assignment 3

Name: Taidgh Murray

Student I.D: 15315901

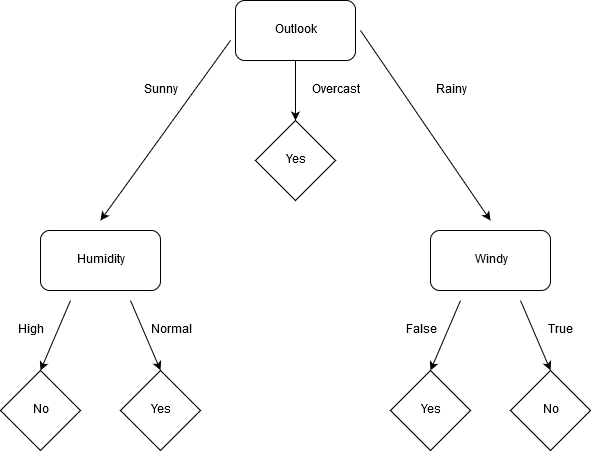
Discipline: College of Science

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.

To implement the CART algorithm.

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
21. """
23. # For reading CSV files & later code
24. **import** random
25. **import** csv

28. # For loading csv files
29. **def** csvLoader(f):
30. f = open(f, "r")
31. l = csv.reader(f)
32. data = list(l)
33. **return** data
35. # Randomly splits 'data' into 'nFolds' amount and creates lists of said splits
36. **def** crossValidationSplit(data, nFolds):
37. dataSplit = list()
38. # dataCopy variable implemented to avoid messy code
39. dataCopy = list(data)
40. foldSize = int(len(data)/nFolds)
41. **for** j **in** range(nFolds):
42. fold = list()
43. **while** len(fold)<foldSize:
44. index = random.randrange(len(dataCopy))
45. fold.append(dataCopy.pop(index))
46. dataSplit.append(fold)
47. **return** dataSplit
49. # Accuracy percentage calculation - predicted value vs actual value
50. **def** accuracyCalculation(predicted, actual):
51. correct = 0
52. **for** k **in** range(len(actual)):
53. **if** actual[k] == predicted [k]:
54. correct+=1
55. **return** ( correct / len(actual) ) \* 100
57. # Evaluates algortihm using the crossValidationSplit function defined above
58. **def** algorithmEvaluation(data, algorithm, nFolds, \*args):
59. folds = crossValidationSplit(data, nFolds)
60. score = list()
61. **for** f **in** folds:
62. trainingSet = list(folds)
63. trainingSet.remove(f)
64. trainingSet = sum(trainingSet, [])
65. testingSet = list()
66. **for** row **in** f:
67. rowCopy = list(row)
68. testingSet.append(rowCopy)
69. rowCopy[-1] = None
70. # Calculates predicition score for given algorithm
71. pred = algorithm(trainingSet, testingSet, \*args)
72. act = [row[-1] **for** row **in** f]
73. accuracy = accuracyCalculation(pred, act)
74. score.append(accuracy)
75. **return** score

78. # Calculate the quality of the data splits
79. **def** splitQuality(groups, classes):
80. nInstances = sum([len(g) **for** g **in** groups])
81. splitQuality = 0
82. **for** g **in** groups:
83. size = len(g)
84. # Avoids division by 0
85. **if** size == 0:
86. **continue**
87. score = 0
88. **for** c **in** classes:
89. p = [row[-1]**for** row **in** g].count(c)/size
90. score += p\*p
91. splitQuality += (1-score)\* (size/nInstances)
92. **return** splitQuality
94. # Splits a dataset based on an attributes
95. **def** testingSplit(index, val, data):
96. left, right = list(), list()
97. **for** r **in** data:
98. **if** r[index] < val:
99. left.append(r)
100. **else**:
101. right.append(r)
102. **return** left, right
104. # Select the best split for the data, by calculating the splitQuality of the data sets
105. **def** getSplit(data):
106. c = list(set(row[-1] **for** row **in** data))
107. splitIndex, splitValue, splitScore, splitGroups = 99999, 99999, 99999, None
108. **for** i **in** range(len(data[0])-1):
109. **for** row **in** data:
110. groups = testingSplit(i, row[i], data)
111. sQ = splitQuality(groups, c)
112. **if** sQ < splitScore:
113. splitIndex, splitValue, splitScore, splitGroups = i, row[i], sQ, groups
114. **return** {'index':splitIndex,'value':splitValue,'groups':splitGroups}
116. # Takes the group of rows assigned to a node and returns the most common value in the group, used to make predictions
117. **def** addToTerminal(group):
118. outcome = [row[-1] **for** row **in** group]
119. **return** max(set(outcome), key = outcome.count)
121. # Create child nodes for the decision tree
122. **def** childNode(node, maxDepth, minSize, depth):
123. left, right = node['groups']
124. **del**(node['groups'])
126. # If no child nodes exist yet
127. **if** **not** left **or** **not** right:
128. node['left'] = node['right'] = addToTerminal(left + right)
129. **return**
131. # If the tree can't get any any larger, but the depht returns a larger value
132. **if** depth >= maxDepth:
133. node['left'], node['right'] = addToTerminal(left), addToTerminal(right)
134. **return**
136. # Left child
137. **if** len(left) <= minSize:
138. # If the left node is smaller than the minimum size, its just added to the tree
139. node['left'] = addToTerminal(left)
140. **else**:
141. # Otherwise it calls the getSplit function on itself
142. node['left'] = getSplit(left)
143. # And recursively calls the function on itself, increasing the depth by 1
144. childNode(node['left'], maxDepth, minSize, depth+1)
146. # Right child
147. **if** len(right) <= minSize:
148. node['right'] = addToTerminal(right)
149. **else**:
150. node['right'] = getSplit(right)
151. childNode(node['right'], maxDepth, minSize, depth+1)
153. # Generated initial decision tree
154. **def** makeDecisionTree(train, maxDepth, minSize):
155. root = getSplit(train)
156. childNode(root, maxDepth, minSize, 1)
157. **return** root
159. # Make prediciton using decision tree
160. **def** prediction(node, row):
161. # If the index node in the row is smaller than the value node
162. **if** row[node['index']] < node['value']:
163. # if the left node is a Python dictionary
164. **if** isinstance(node['left'], dict):
165. # Recursively calls the prediction function using the left node and the row
166. **return** prediction(node['left'], row)
167. **else**:
168. # Otherwise just returns the left node
169. **return** node['left']
171. **else**:
172. **if** isinstance(node['right'], dict):
173. **return** prediction(node['right'], row)
174. **else**:
175. **return** node['right']

178. # Calling the CART algorithm
179. **def** cart(train, test, maxDepth, minSize):
180. # Defines a tree using the makeDecisionTree funcion
181. tree = makeDecisionTree(train, maxDepth, minSize)
182. # Initialises empty list, called 'predictions' to hold predictions
183. predictions = list()
184. # Fills prediction list with predictions for each row of info in the test data
185. **for** row **in** test:
186. p = prediction(tree, row)
187. predictions.append(p)
188. **return** predictions
190. # Set Random seed
191. random.seed(7)
193. # Allows for some user input, they can chose a different file to be tested, and change the amount of folds, the maximum depth, and the minimum size of the tree
195. # Load data
196. file = (input("Please enter a file, or leave blank for owls.csv: ") **or** 'owls.csv')
197. data = csvLoader(file)
199. # Evaluate algorithm
200. nFolds = (input("Enter the number of folds you wish to create, or leave blank for default (3):") **or** 3)
201. maxDepth = (input("Enter the maximum depth of the tree, or leave blank for default (5):") **or** 5)
202. minSize = (input("Enter the number of folds you wish to create, or leave blank for default (10):") **or** 10)
203. scores = algorithmEvaluation(data, cart, nFolds, maxDepth, minSize)
205. # Formats results
206. **print**('Scores: {}'.format(scores))
207. # Mean accuracy of accuracy score for CART Algorithm
208. **print**('Mean Accuracy: %.2f%%' % (sum(scores)/(len(scores))))
210. # Users can then read the completed data before closing the program
211. input()