Course Code: CT4101 (Machine Learning) Assignment: 2 Name: *Seshadri Sundarrajan* Course Code: 1CSD1 Student ID: 19230107

# Random Forest From Scratch - REPORT (Dataset:hazzlenuts)

* Implemented Random Forest algorithm encompassing Bootstrap sampling and random feature subspace. Each classifier(tree) of the algorithm was built on C4.5 algorithm.
* Random forest is an ensemble technique and known for better accuracy as it combines output from several weak learners to form base classifier. Moreover, once Decision tree classifier is implemented it requires minimal effort to build bootstrap aggregation methodology on top of it to get bagged model.

# pROGRAMMING LANGUAGE AND LibrarIES used

* **Python** 
  + Primarily used *Pandas* Data frame to store tree information (Meta data).
  + For computations made use of Numpy Arrays.
  + Matplotlib is used for visualization ROC curves.
  + Networkx is used for decision tree visualization.

# Data Pre-processing (Done in EXCEL)

* The provided dataset had columns representing instances. To feed into Pandas data frame and for ease of interpreting *transposed* the data to get instances along the row.
* ‘Sample id’ column has no impact on our classifier as it’s just a key, so dropped it.

# dESIGN logic

### **Random Forest**

Algorithm is built upon 2 random processes

1. *Bootstrap sampling*
   * At each tree, input data is split to n-samples with repetition.
2. *Feature Subspace*
   * At each tree, features are subset randomly

**Parameters Considered for design**: These are part of constructor (should be passed while creating object)

|  |  |
| --- | --- |
| **Parameters** | **Description** |
| Total\_Trees | Number of decision trees to be constructed |
| Tree Depth | Maximum depth of a decision tree |
| Bootstrap size | Bootstrap sample size per tree |
| Features\_per\_tree | Random number of features on which a tree is built |
| Random\_state | Random seed |

**Training.**

Each individual tree is constructed using C4.5 algorithm implemented separately. As part of C4.5 design we get meta information on each tree and we store it.

**Testing**

For each test data, we get the output from each tree. Finally, output of a single test data will be aggregation of all the collected tree outputs. (Mode).

**Convergence**: All trees are trained and individual tree convergence depends on Decision Tree convergence.

**Methods Available:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Parameters** | **Return** | **Description** |
| Fit() | - | - | Starts building model. |
| Predict() | Array (x\_test) | Predictions(array) | Predicts output class for each element in the input array from the built model |
| Predict\_prob() | Array (x\_test) | Output class probabilities (array) | Returns class probabilities (Mean probability across all tress for each class) - To get ROC Curves |
| Write\_meta\_data() | File\_name | Excel workbook | Writes each tree meta info to a sheet. |

**NOTE**: Private method named ‘**Predict\_tree\_op**’ is called inside ‘Predict’ to get output class from an individual tree for given test data. So, can we can perform aggregation on top of all tree outputs.

### **c4.5**

Algorithm is built entirely to support ***continuous*** features and uses ***entropy*** as impurity measure.

**Binning of continuous features**

* For given feature, set of thresholds are chosen for evaluation by taking mean between successive values arranged in ascending order. (For n values we get n-1 threshold).
* For each threshold we calculate its entropy and overall entropy based on class proportions.
* Finally, best threshold that has minimum overall entropy is chosen to get homogeneous splits with high information gain. We split data into 2 partitions namely (<=Threshold & >Threshold).

**Meta data driven Approach**

1. We select the best feature that gives us homogenous split and start navigating down.
2. After the first split we store the corresponding information into our meta data frame.
3. Then next corresponding split takes place by accessing the subset of data that branched out and selecting the necessary feature and its threshold from the meta info.
4. Thus, each split (except first split) is dependent on the meta data of previous split.
5. This process keeps reoccurring until convergence.

## **Meta Data Frame Structure**

***‘Split condition’*** is list of strings (stored cummulatively at each node depending on its branch hierarchy). Ex: - [‘shell radius <= 5.012’,’width<=14.148’], this will drive us to get the subset needed for next split.

**‘Next split feature’** is a list with feature names upon which successive splits can be done. They are decided in such a way that they do not repeat the same feature that branched out to the current node.

**Recursive Logic**

1. At each non-leaf node, we get the node’s split condition and filter the data subset for next split.
2. Then from ***‘next\_split\_features’*** find best feature and its threshold along with overall entropy.
3. Then we populate child nodes data along with output class probabilities and leaf flag**(‘leaf\_f’**).

**Convergence**

* If a node is pure then we label them as ‘leaf’ in our meta data, so that no further split occurs.
* On contrary, if our tree depth is reached and yet node is not pure, again we label them as ‘leaf’.

**Parameters Considered for design:** Tree depth

**Methods Available:** Fit() [ Starts building the tree and returns the tree meta information as data frame].

**NOTE**: Private methods named *‘tree\_initial\_split’, ‘build\_tree’* and *‘recursive\_tree’* are used to build the tree and return tree information.

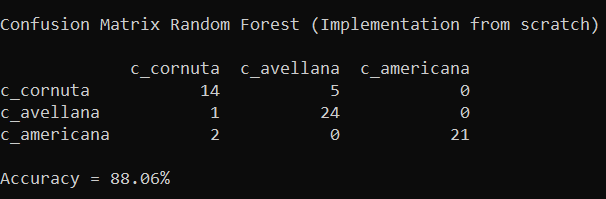
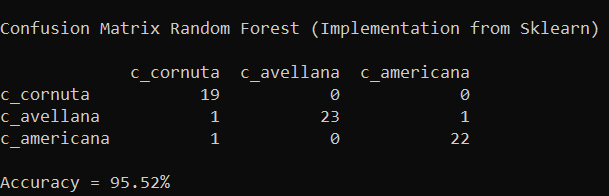
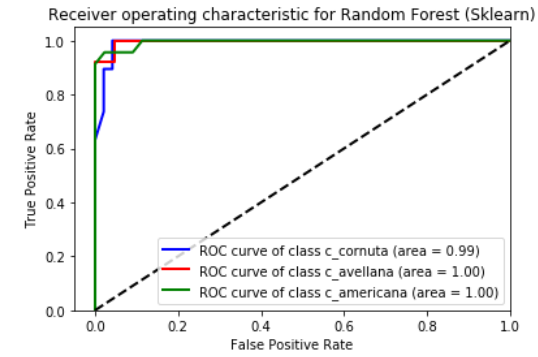
## **decision tree visualized from meta data** (PACKAGE USED: NETWORKX)

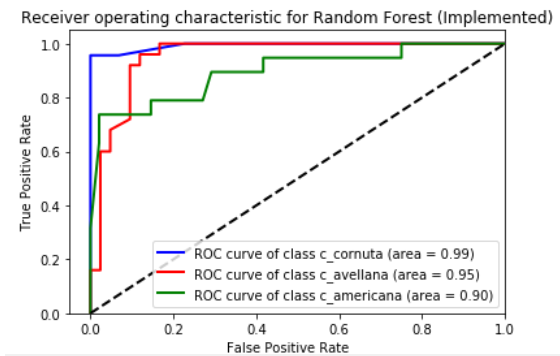
# RANDOM FOREST (sCI-KIT LEARN)

Created model from ML library to replicate the same behaviour as that of implementation done, where test and train data are split in ratio 1:2.

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Scratch Implementation** | **Sklearn library** |
| Total\_Trees | 11 | 11 |
| Tree Depth | 5 | 5 |
| Bootstrap size | 100 | Not provided |
| Features\_per\_tree | 6 | Not provided |
| Random\_state | 10 | 10 |

# MODEL COMPARISON

* Sklearn Model execution **time** was much faster than the implementation. Potential reasons might be caching and parallel processing.
* Initial comparison was done at random state =10 and **confusion matrices** obtained are displayed.
* Mean accuracy after 10 random splits is ***90% for sklearn*** random forest and is **88% for implementation** done.
* ROC cures for both the models are generated at random state = 10 and we observe that area under the curve for sklearn random forest is almost 100% for all the 3 classes but in implementation done we have 99% for ‘c\_cornuta’ and 90% for ‘c\_americana’ and 95% for ‘c\_avellana’. Thus, model implemented tend to misclassify last 2 classes.



* Another important difference between 2 implementations is that there is no interface in sklearn random forest to specify bootstrap sample and feature size per tree, as it is taken care dynamically.
* On comparison to Assignment 1, here haven’t removed any correlated features because while generating random feature subset we get a chance of picking correlated feature while its dependent is left out, so our predictions might improve.
* To sum up, scratch implementation could meet ML library performance if it is tuned better with extra parameters and if we calculate bootstrap sample and feature size per tree dynamically from data based on proven methods and taking in to account dynamic programming to compensate time delay. Overall, **Sklearn Random Forrest has better performance**.

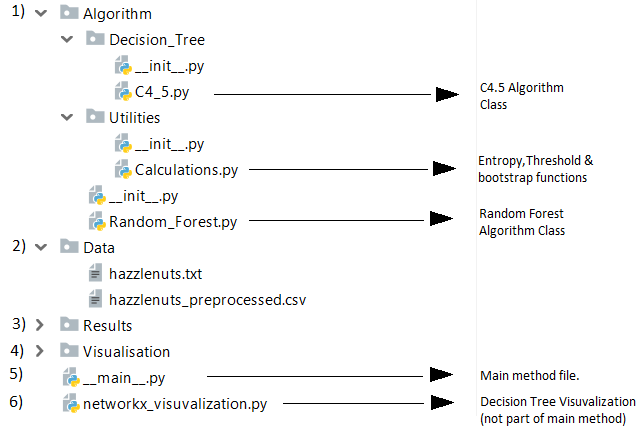
# REFERENCES

1. <https://www.sebastian-mantey.com/posts/random-forest-algorithm>
2. <https://books.google.ie/books?id=HExncpjbYroC&printsec=frontcover&source=gbs_ge_summary_r&redir_esc=y#v=onepage&q&f=false>
3. <https://www.sebastian-mantey.com/posts/decision-tree-algorithm-part-2-entropy>
4. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
5. <https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html>

# Appendix – SOURCE CODE

**NOTE**: Runnable file is ‘***\_\_main\_\_.py*** ‘(the main method file) and it makes the call to algorithm.

### **Package Structure**



### **Random\_Forest.py**

1. **import** random
2. **import** pandas as pd
3. **import** numpy as np
4. **import** pprint
5. **from** operator **import** itemgetter
7. # Files for this below 2 imports are attached
9. **from** Algorithm.Decision\_Tree.C4\_5 **import** \* # CART implementation from scratch
10. **from** Algorithm.Utilities.Calculations **import** \*   # All helper functions involving calculations



15. **class** random\_forest():
17. # At each tree we store features sent and their indexes which will be used for testing
18. features\_idx\_subseted = []
19. feature\_subseted = []
21. # We store Meta info in a dictionary with assumed structure { 'Tree\_1' : 'Meta\_Data\_frame\_of\_tree1',.....}
22. meta\_dfs = {}
23. **def** \_\_init\_\_ (self,data,total\_trees,bootstrap\_size,feature\_per\_tree,tree\_depth = 5,random\_seed = 10):
25. self.feature\_per\_tree = feature\_per\_tree
26. self.bootstrap\_size = bootstrap\_size
27. self.total\_trees = total\_trees
28. self.data = data
29. self.features =data.columns.tolist()[0:-1]
30. self.lable\_name = data.columns.tolist()[-1]
31. self.tree\_depth = tree\_depth
32. self.random\_seed = random\_seed

35. **def** fit(self):
37. '''''Build each tree using Bootstrap sample Random Feature subspace.'''
39. random.seed(self.random\_seed)
41. **for** tree **in** range(self.total\_trees):
42. sample = get\_bootstrap(self.data,self.bootstrap\_size,self.random\_seed)
44. sub\_feature = random.sample(range(len(self.features)), k=self.feature\_per\_tree)
45. random\_forest.features\_idx\_subseted.append(sub\_feature)
47. feature\_subset = list(itemgetter(\*sub\_feature)(self.features))
48. random\_forest.feature\_subseted.append(feature\_subset)
50. # Call to decision tree
52. c = DT\_classifier(self.tree\_depth,sample,feature\_subset,self.lable\_name,sample[self.lable\_name].unique().tolist())
54. c.fit()

57. # Appending decision tree meta info of this tree to our random forest meta dictionary
58. random\_forest.meta\_dfs[tree] = c.mdf

61. **def** \_\_predict\_tree\_op(self,ip\_arr,cols,mdf,get\_prob = 'N'):
63. '''''Predicts lables for given tree for input array of attributes.
65. Parameters:
66. Attributes Array,Meta Data frame
68. Summary:
69. -Gets all Leaf Node conditions from Meta data frame
70. -Checks which condition is met by test data and maps leaf node pertaining to the condition.
71. -Predicts the most probable output class the probabilities in leaf node.
73. Returns:
74. Predicted lables for the tree passed'''
76. op\_lst = []
77. leaf\_node\_df = mdf [mdf['leaf\_f'] == 'Y']
78. leaf\_conditions = leaf\_node\_df['split\_condition']
79. **for** elm **in** ip\_arr:
80. p = pd.Series(elm,index = cols)
81. **for** num,condition **in** enumerate(leaf\_conditions):
82. check = []
83. **for** cond **in** condition:
84. col = cond.split()[0]
85. val = ' '.join(cond.split()[1:])
86. **if**  eval(str(p[col]) + val):
87. check.append('True')
88. **else**:
89. check.append('False')
90. **if** 'False' **not** **in** check:
91. class\_prob\_df = leaf\_node\_df.iloc[num,][self.data[self.lable\_name].unique().tolist()]
92. **if** get\_prob == 'Y':
93. op = class\_prob\_df.values.tolist()
94. op\_lst.append(op)
95. **else**:
96. op = class\_prob\_df[class\_prob\_df == class\_prob\_df.max()].index.values #returns column name with max value
97. op\_lst.append(op[0])
98. **break**
100. **return** np.array(op\_lst)

103. **def** predict(self,ip\_arr):
105. '''''Predicts lables from output lables obtained from each tree.
107. Parameters:
108. Attributes Array,
110. Returns:
111. Predicted lables'''
113. votes\_arr = []
115. **for** tree **in** range(self.total\_trees):
117. cols = random\_forest.feature\_subseted[tree]
118. idx\_cols = random\_forest.features\_idx\_subseted[tree]
120. idx\_data =np.array([list(arr[idx\_cols]) **for** arr **in** ip\_arr])
122. op = self.\_\_predict\_tree\_op(idx\_data,cols,random\_forest.meta\_dfs[tree])
123. votes\_arr.append(op)
124. odf = pd.DataFrame(np.array(votes\_arr))
125. **return** odf.mode().iloc[0].values
127. **def** write\_meta\_data(self,file\_name):
129. with pd.ExcelWriter(file\_name) as workbook:
130. **for** tree,mdf **in** random\_forest.meta\_dfs.items():
131. mdf.to\_excel(workbook, sheet\_name='Tree\_'+str(tree+1),index=False)
132. **print**("\nCheck '{}' for meta information at each tree.".format(file\_name))




138. **def** predict\_prob(self,ip\_arr):
140. '''''Predicts lables from output lables obtained from each tree.
142. Parameters:
143. Attributes Array,
145. Returns:
146. Predicted lables'''
148. votes\_arr = []
150. **for** tree **in** range(self.total\_trees):
152. cols = random\_forest.feature\_subseted[tree]
153. idx\_cols = random\_forest.features\_idx\_subseted[tree]
155. idx\_data =np.array([list(arr[idx\_cols]) **for** arr **in** ip\_arr])
157. op = self.\_\_predict\_tree\_op(idx\_data,cols,random\_forest.meta\_dfs[tree],get\_prob ='Y')
158. votes\_arr.append(op)
159. prob = np.array(votes\_arr)
160. **return** np.mean(prob,axis=0)

### **C4\_5.py**

1. **import** pandas as pd
2. **import** numpy as np
4. **from** Algorithm.Utilities.Calculations **import** \*

7. **class** DT\_classifier():
9. # Meta Data Frame to store decision tree information
10. mdf  = pd.DataFrame()
12. **def** \_\_init\_\_(self,max\_depth,train\_df,feature\_list,lable\_name,lable\_list):
13. self.max\_depth = max\_depth
14. self.train\_df = train\_df
15. self.feature\_list = feature\_list
16. self.lable\_name = lable\_name
17. self.lable\_list = lable\_list

20. **def** \_\_tree\_initial\_split(self,feature\_list,lable,df\_test):
22. """Initiate initial tree split to get data into meta df for further spliting.
24. Parameters -
26. Attributes,Label name,Data
28. Returns -
30. Meta Data Frame
31. """
32. split\_feature,threshold,overall\_entropy = discretize(df\_test,lable,feature\_list)
33. cols = ['node\_number','parent\_node','split\_feature','split\_category','split\_condition','overall\_entropy','leaf\_f','next\_split\_features']
34. val = [['1',np.nan,split\_feature,np.nan,[threshold],overall\_entropy,'N']]
35. feature\_for\_next\_split = feature\_list.copy()
36. feature\_for\_next\_split.remove(split\_feature)  # removing current split feature for further spliting
37. val[0].append(feature\_for\_next\_split)
38. lables = df\_test[lable].unique().tolist()
39. **for** i **in** lables:
40. cols.append(str(i))   # creating columns as per no of labels in op class
41. class\_data = len(df\_test[df\_test[lable] == i])
42. total\_data = len(df\_test)
43. val[0].append(class\_data/total\_data) # probability for that lable in the node
44. meta\_df = pd.DataFrame(val,columns=cols)
45. #for num,levels in enumerate(df\_test[split\_feature].unique()): #Branches from the first node
46. **for** num,operator **in** enumerate([' <= ',' > ']):
47. condition = operator +str(threshold)
48. vals = [['1\_'+str(num+1),'1',split\_feature,condition,[split\_feature+condition],np.nan,np.nan,feature\_for\_next\_split]]
49. **for** lable **in** lables: vals[0].append(np.nan) #appending values for op class probability
50. meta\_df = meta\_df.append(pd.DataFrame(vals,columns=cols),ignore\_index=True)[cols] # adding observations to meta\_df
51. **return** meta\_df




57. **def** \_\_build\_tree(self,data,meta\_df,lable\_name,lable\_list,final\_split = 'N'):
59. '''''Split tree from the provided state to next state.
61. Parameters:
63. Data,Meta Data Frame,Label name,Classes in lable,Final split flag
65. Summary:
67. -Filters the nodes that are not split using meta info
68. -Filtered Nodes which are non-terminal will be split further.
69. -Filtered Nodes are updated with Leaf flag,overall entropy and class probability information in meta data frame.
70. -If final split flag = 'Y', only weights are updated no further spliting (If tree\_depth has reached but node is not leaf then we do this as last step)
72. Returns:
73. Temporary Data Frame containg meta information of new split done'''
75. df\_node\_to\_split = meta\_df[(meta\_df['parent\_node'].notnull()) & (pd.isnull(meta\_df['overall\_entropy'])) ]
76. cols = meta\_df.columns.tolist()
77. temp\_df = pd.DataFrame(columns=cols)
78. **for** i **in** range(0,len(df\_node\_to\_split)):
79. leaf = False
80. series = df\_node\_to\_split.iloc[i,]
81. node\_number,features\_for\_split,split\_condition = series['node\_number'],series['next\_split\_features'],series['split\_condition']
83. #Filtering data subset based on split condition dict
84. subset = data
85. **for** cond **in** split\_condition:
86. key = cond.split()[0]
87. value = ' '.join(cond.split()[1:])
88. subset = eval('subset[subset[key]'+ value+']')
90. probs = []
91. #Updating class label  propbailites and lead flag
92. **for** lable **in** lable\_list:
93. **if** len(subset) >0: # for zreo division error
94. prob = len(subset[subset[lable\_name] == lable]) / len(subset)
95. meta\_df.loc[meta\_df['node\_number'] ==series['node\_number'],str(lable)] = prob
96. probs.append(prob)
97. **if** (1 **in** probs) **or** (final\_split == 'Y') **or** (len(features\_for\_split) == 0):
98. meta\_df.loc[meta\_df['node\_number'] ==series['node\_number'],'leaf\_f'] = 'Y'
99. meta\_df.loc[meta\_df['node\_number'] ==series['node\_number'],'overall\_entropy'] = 0
100. leaf = True
101. **else**:
102. meta\_df.loc[meta\_df['node\_number'] ==series['node\_number'],'leaf\_f'] = 'N'
104. **if** leaf **or** (final\_split == 'Y') :
105. **continue**
107. #Get split feature for further split if its not a leaf node
109. split\_feature,threshold,overall\_entropy = discretize(subset,lable\_name,features\_for\_split)
110. meta\_df.loc[meta\_df['node\_number'] ==series['node\_number'],'overall\_entropy'] = overall\_entropy #Weighted entropy update
111. next\_features = features\_for\_split.copy()
112. next\_features.remove(split\_feature) # child's next feature for split
114. #child observations entry
115. **for** num,operator **in** enumerate([' <= ',' > ']):
116. #for num,levels in enumerate(subset[split\_feature].unique()):
117. condition = split\_condition.copy()
118. condition.append(split\_feature + operator + str(threshold))
119. vals = [[node\_number+'\_'+str(num+1),node\_number,split\_feature,operator + str(threshold),condition,np.nan,np.nan,next\_features]]
120. **for** lbl **in** lable\_list: vals[0].append(np.nan)
121. temp\_df = temp\_df.append(pd.DataFrame(vals,columns=cols),ignore\_index=True)[cols]
122. **return** temp\_df
124. **def** \_\_recursive\_tree(self,data,max\_depth,lable\_name,feature\_list,lable\_list):
126. '''''Initiates tree building and splits child nodes until convergence.
128. Parameters:
129. Data,Tree Depth,Label name,Data,Attributes,Output class names
131. Returns:
132. Meta Data Frame'''
134. meta\_df = self.\_\_tree\_initial\_split (feature\_list,lable\_name,data)
135. max\_depth = max\_depth - 1 #as already 1 split has been done by us
136. **for** i **in** range(0,max\_depth):
137. temp\_df = self.\_\_build\_tree(data,meta\_df,lable\_name,lable\_list)
138. meta\_df = meta\_df.append(temp\_df,ignore\_index=True)[meta\_df.columns.tolist()]
139. temp\_df = self.\_\_build\_tree(data,meta\_df,lable\_name,lable\_list,final\_split = 'Y') # just to update meta data no split occurs here
140. meta\_df = meta\_df.append(temp\_df,ignore\_index=True)[meta\_df.columns.tolist()]
141. **return** meta\_df



146. **def** fit(self):
147. df = self.\_\_recursive\_tree(self.train\_df,self.max\_depth,self.lable\_name,self.feature\_list,self.lable\_list)
148. DT\_classifier.mdf = df

### **Calculations.py**

1. **import** numpy as np
3. **def** get\_threshold(df,col,lable):
5. '''''Get best threshold value for the attribute
7. Parameters :
8. Data frame,Attribute,Lable name
10. Returns:
11. Entropy value,Threshold value'''
13. df = df[[col,lable]]
14. uniq\_vals = df[col].unique().tolist()
15. uniq\_vals.sort()
16. thresholds = [(uniq\_vals[idx]+uniq\_vals[idx+1])/2 **for** idx **in** range(0,len(uniq\_vals) -1)]
17. weighted\_entropy = []
19. **for** threshold **in** thresholds:
20. left = df[df[col] <= threshold][lable]
21. right = df[df[col] > threshold][lable]
22. entropy = calc\_wgtd\_entropy\_numeric(left,right)
23. weighted\_entropy.append(entropy)
24. **if** thresholds == []:
25. weighted\_entropy = [0]
26. thresholds = [0]
28. **return** [min(weighted\_entropy),thresholds[weighted\_entropy.index(min(weighted\_entropy))]]


32. **def** calc\_wgtd\_entropy\_numeric( left, right):
34. '''''Get overall entropy for an attribute
36. Parameters :
37. Data towards left and right of threshold (i.e, <= threshold & > threshold )
39. Returns:
40. Overall entropy'''
42. total\_elements = len(left) + len(right)
43. ent\_left = entropy(left)
44. ent\_right = entropy(right)
45. weighted\_entropy = ((len(left) / total\_elements) \* ent\_left) + ((len(right) / total\_elements) \* ent\_right)
46. **return** weighted\_entropy

49. **def** entropy(df):
51. '''''Get entropy for a subset
53. Parameters :
54. Lable values of subset
56. Returns:
57. Entropy'''
59. op\_class, count = np.unique(df.values, return\_counts=True)
60. entropy = np.sum([(-count[i] / np.sum(count)) \* np.log2(count[i] / np.sum(count)) **for** i **in** range(len(op\_class))])
61. **return** entropy


65. **def** discretize(df,lable,feature\_list):
67. '''''Discretization (binning) of continuous data
69. Parameters :
70. Data Frame,Lable Name,Attributes
72. Returns:
73. Attribute,Threshold,Overall Entropy'''
75. val = []
76. d = {}
77. **for** i **in** feature\_list:
78. op = get\_threshold(df,i,lable)
79. val.append(op)
80. d[op[1]] = i
82. ent = [value[0] **for** value **in** val]
83. thr = [value[1] **for** value **in** val]
84. **return** d[thr[ent.index(min(ent))]],thr[ent.index(min(ent))],min(ent)

87. **def** get\_bootstrap(data, bootstrap\_size,random\_seed):
89. ''''' Bootstraping for random forrest
91. Parameters :
92. Data Frame, Bootstarp size, Random state
94. Returns:
95. Boostrap sample'''
97. np.random.seed(random\_seed)
98. bootstraps = np.random.randint(low=0, high=len(data), size=bootstrap\_size)
99. df\_bootstrap = data.iloc[bootstraps]
100. **return** df\_bootstrap

### **\_\_main\_\_.py**

1. **import** random
2. **import** pandas as pd
3. **import** numpy as np
4. **from** Algorithm.Random\_Forest **import** random\_forest
5. **from** sklearn.ensemble **import** RandomForestClassifier
6. **from** sklearn.metrics **import** confusion\_matrix
7. **from** sklearn.model\_selection **import** train\_test\_split
8. **from** sklearn **import** metrics
9. **from** sklearn.model\_selection **import** cross\_val\_score,cross\_val\_predict
11. df = pd.read\_csv('Data\\hazzlenuts\_preprocessed.csv')
13. features = df.columns[0:-1].tolist()
14. output\_lable\_name = df.columns[-1]

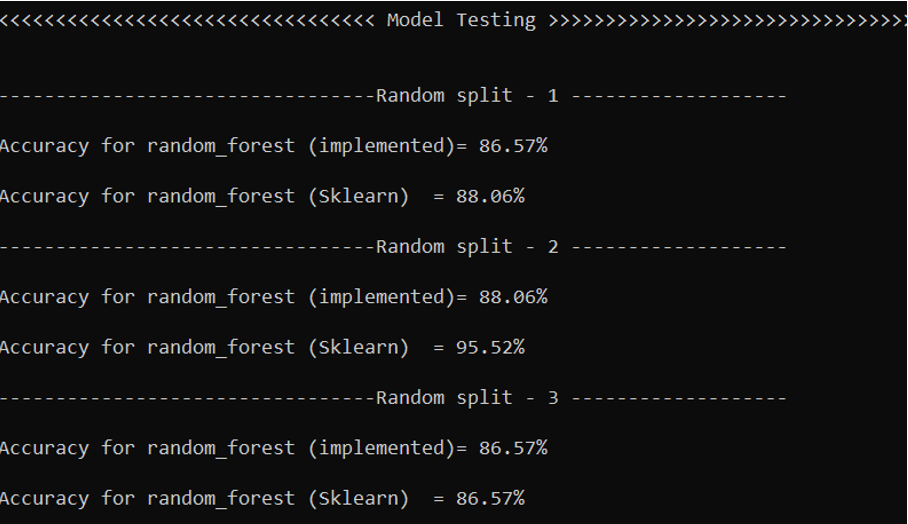
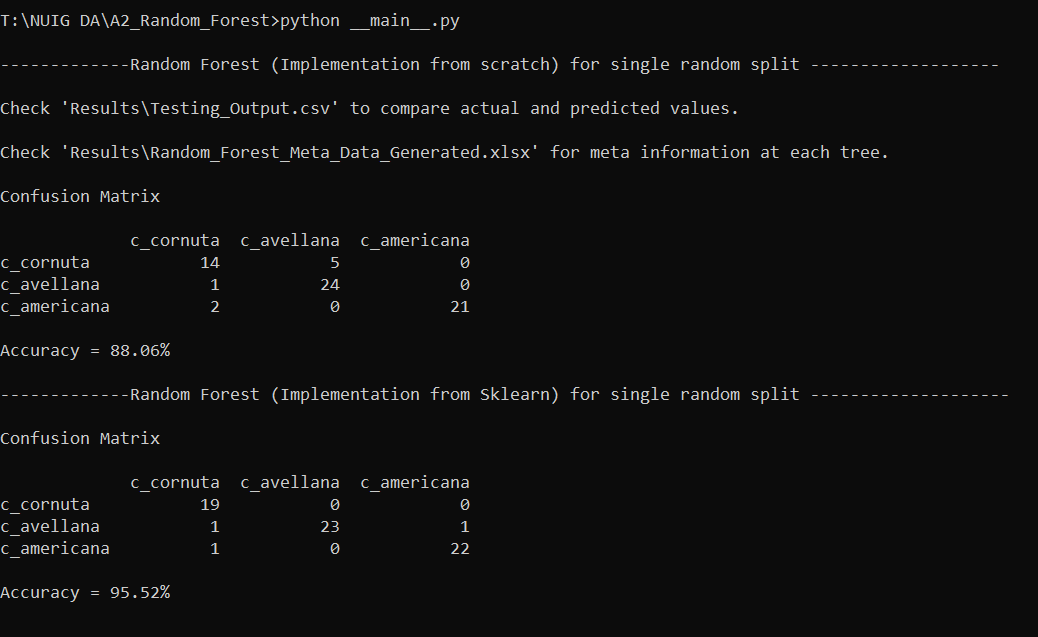
17. x = df[features].values
18. y = df[output\_lable\_name].values
20. x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size = 0.33, random\_state = 1)
22. train\_df = pd.DataFrame(x\_train,columns = features)
23. train\_df[output\_lable\_name] = y\_train
24. train\_df
26. **def** gen\_random\_splits(n):
27. sk\_rf\_acc = []
28. rf\_acc = []
29. **for** split **in** range(0,n):
31. **print**('\n---------------------------------Random split - {} -------------------'.format(split+1))
32. x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size = 0.33, random\_state = split)
34. train\_df = pd.DataFrame(x\_train,columns = features)
35. train\_df[output\_lable\_name] = y\_train
36. train\_df
38. r = random\_forest(data = train\_df,total\_trees = 11,tree\_depth = 5,random\_seed = 10,feature\_per\_tree = 6,bootstrap\_size=100)
39. r.fit()
40. op= r.predict(x\_test)
41. acc\_rf = metrics.accuracy\_score(y\_test, op)
42. **print**('\nAccuracy for random\_forest (implemented)= {}%'.format(round(acc\_rf\*100,2)))
43. rf\_acc.append(acc\_rf)
45. classifier = RandomForestClassifier(n\_estimators = 11,max\_features = 6, criterion = 'entropy', random\_state = 10,max\_depth = 5)
46. classifier.fit(x\_train, y\_train)
47. op = classifier.predict(x\_test)
48. acc\_sk = metrics.accuracy\_score(y\_test, op)
49. **print**('\nAccuracy for random\_forest (Sklearn)  = {}%'.format(round(acc\_sk\*100,2)))
50. sk\_rf\_acc.append(acc\_sk)
52. **print**('\n------------------------------------------------------')
53. **print**('\nMean Accuracy for random\_forest (Implemented) after {} random splits  = {}%'.format(n,round(np.array(rf\_acc).mean()\*100,3)))
54. **print**('\nMean Accuracy for random\_forest (Sklearn) after {} random splits  = {}%'.format(n,round(np.array(sk\_rf\_acc).mean()\*100,3)))
56. **if** \_\_name\_\_ == "\_\_main\_\_":
58. **print**('\n-------------Random Forest (Implementation from scratch) for single random split -------------------')
59. r = random\_forest(data = train\_df,total\_trees = 11,tree\_depth = 5,random\_seed = 10,feature\_per\_tree = 6,bootstrap\_size=100)
61. r.fit()
62. op= r.predict(x\_test)
63. prob\_arr = r.predict\_prob(x\_test)
64. pd.DataFrame({'Actual': y\_test,'Predicted':op}).to\_csv('Results\\Testing\_Output.csv',index=False)
66. **print**("\nCheck 'Results\\Testing\_Output.csv' to compare actual and predicted values.")
67. r.write\_meta\_data('Results\\Random\_Forest\_Meta\_Data\_Generated.xlsx')
69. **print**('\nConfusion Matrix \n')
70. cm = confusion\_matrix(y\_test, op)
71. **print**(pd.DataFrame(cm,columns=train\_df.iloc[:,-1].unique(),index = train\_df.iloc[:,-1].unique() ))

74. acc = metrics.accuracy\_score(y\_test, op)
75. **print**('\nAccuracy = {}%\n'.format(round(acc\*100,2)))
77. **print**('-------------Random Forest (Implementation from Sklearn) for single random split --------------------')

80. classifier = RandomForestClassifier(n\_estimators = 11,max\_features = 6, criterion = 'entropy', random\_state = 10,max\_depth = 5)
81. classifier.fit(x\_train, y\_train)
82. op = classifier.predict(x\_test)
84. **print**('\nConfusion Matrix\n')
85. cm\_sk = confusion\_matrix(y\_test, op)
86. **print**(pd.DataFrame(cm\_sk,columns=train\_df.iloc[:,-1].unique(),index = train\_df.iloc[:,-1].unique() ))
88. acc\_sk = metrics.accuracy\_score(y\_test, op)
89. **print**('\nAccuracy = {}%'.format(round(acc\_sk\*100,2)))

92. **print**('\n\n<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Model Testing >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n')
93. gen\_random\_splits(10)

### **Console Output:**

Executing \_\_main\_\_.py file inside compressed zip.(Total execution time = 2m:45s [Inluding 10 random splits])

