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.

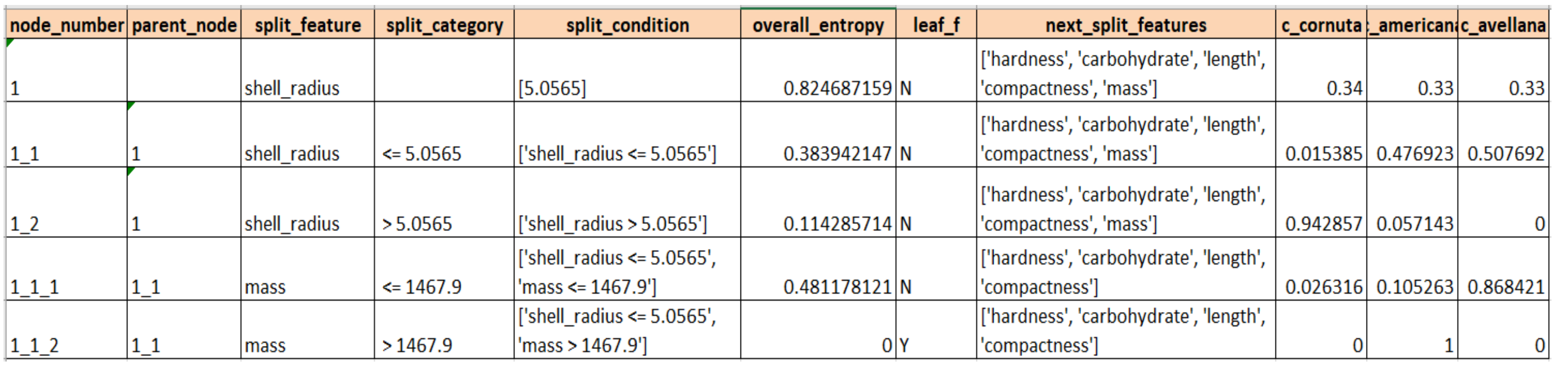
Binning of continuous features

- 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



node_number	parent_node	split_feature	split_category	split_condition	overall_entropy	leaf_f	next_split_features	c_cornuta	_americana	c_avellana
1		shell_radius		[5.0565]	0.824687159	N	['hardness', 'carbohydrate', 'length', 'compactness', 'mass']	0.34	0.33	0.33
1_1	1	shell_radius	<= 5.0565	['shell_radius <= 5.0565']	0.383942147	N	['hardness', 'carbohydrate', 'length', 'compactness', 'mass']	0.015385	0.476923	0.507692
1_2	1	shell_radius	> 5.0565	['shell_radius > 5.0565']	0.114285714	N	['hardness', 'carbohydrate', 'length', 'compactness', 'mass']	0.942857	0.057143	0
1_1_1	1_1	mass	<= 1467.9	['shell_radius <= 5.0565', 'mass <= 1467.9']	0.481178121	N	['hardness', 'carbohydrate', 'length', 'compactness']	0.026316	0.105263	0.868421
1_1_2	1_1	mass	> 1467.9	['shell_radius <= 5.0565', 'mass > 1467.9']	0	Y	['hardness', 'carbohydrate', 'length', 'compactness']	0	1	0

'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

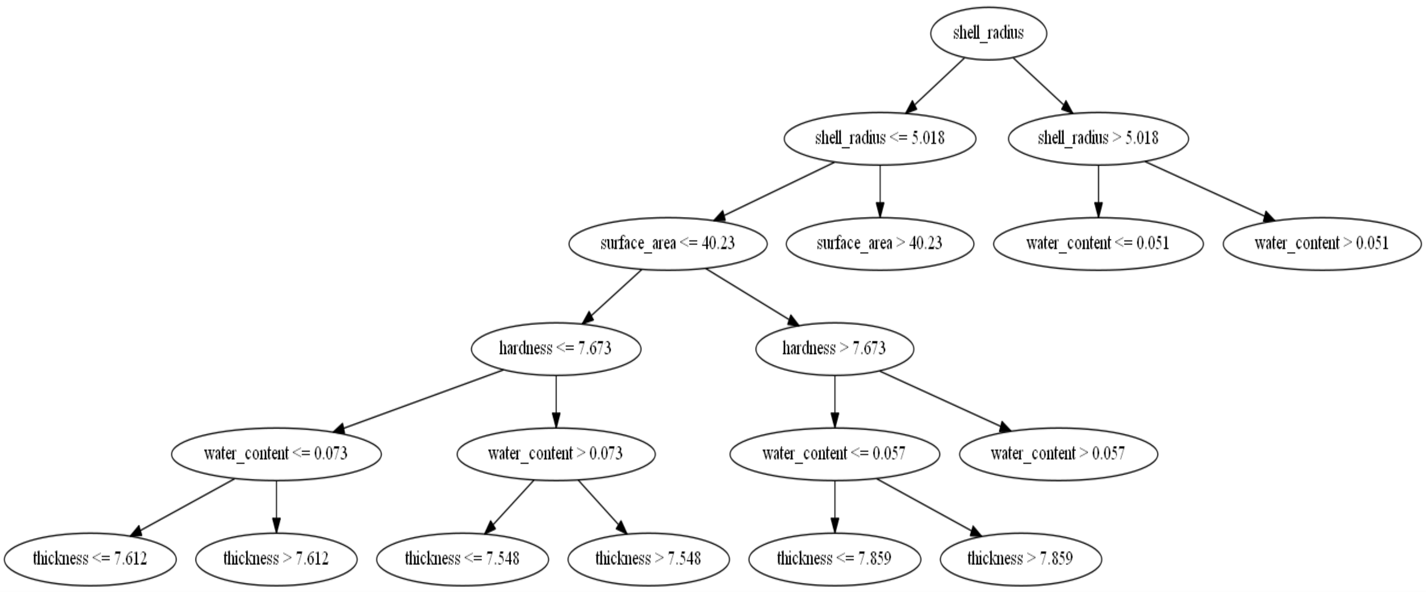
- If a node is pure then we label them as 'leaf' in our meta data, so that no further split occurs.
- On contrary, if our tree depth is reached and yet node is not pure, again we label them as 'leaf'.

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)



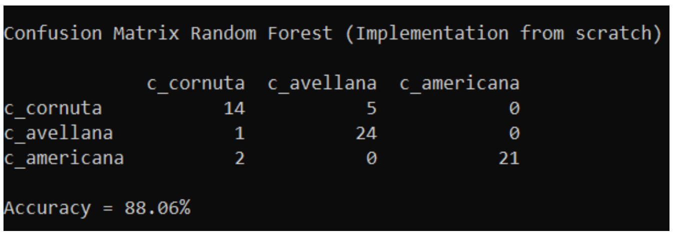
RANDOM FOREST (SCI-KIT LEARN)

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.

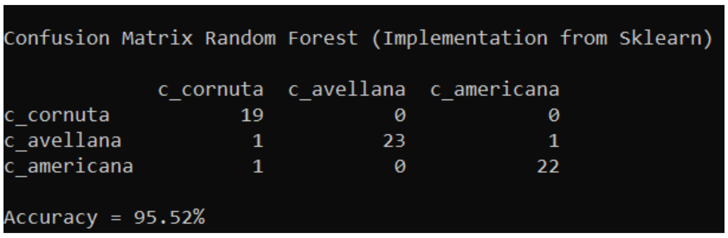
Parameters	**Scratch Implementation**	**Sklearn library**
Total_Trees	11	11
Tree Depth	5	5
Bootstrap size	100	Not provided
Features_per_tree	6	Not provided
Random_state	10	10

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.

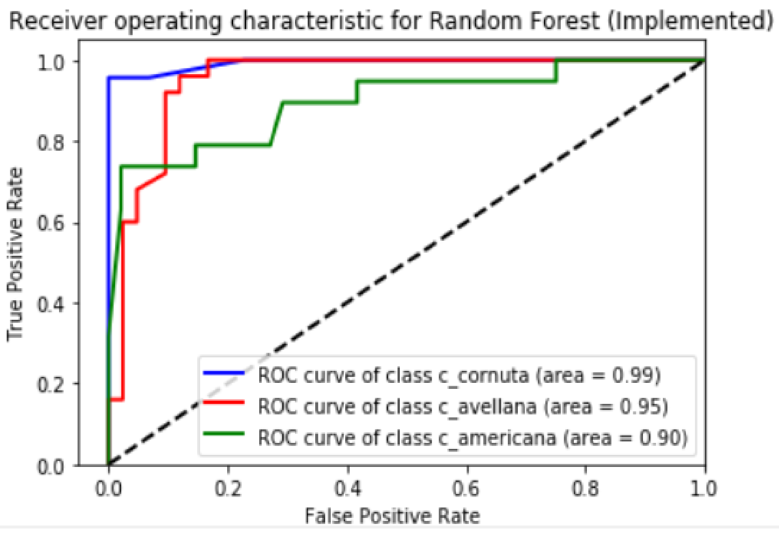


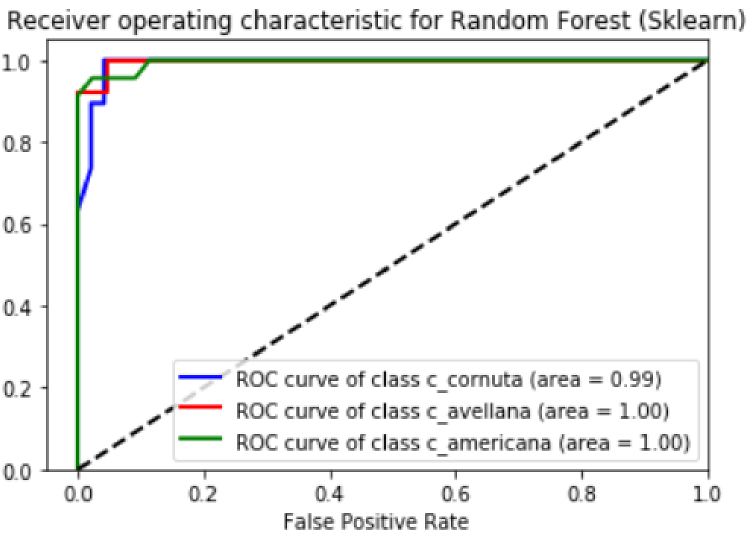
```
Confusion Matrix Random Forest (Implementation from scratch)

             c_cornuta  c_avellana  c_americana
c_cornuta           14           5            0
c_avellana           1          24            0
c_americana          2           0           21

Accuracy = 88.06%
```



```
Confusion Matrix Random Forest (Implementation from Sklearn)

             c_cornuta  c_avellana  c_americana
c_cornuta           19           0            0
c_avellana           1          23            1
c_americana          1           0           22

Accuracy = 95.52%
```

- 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**.

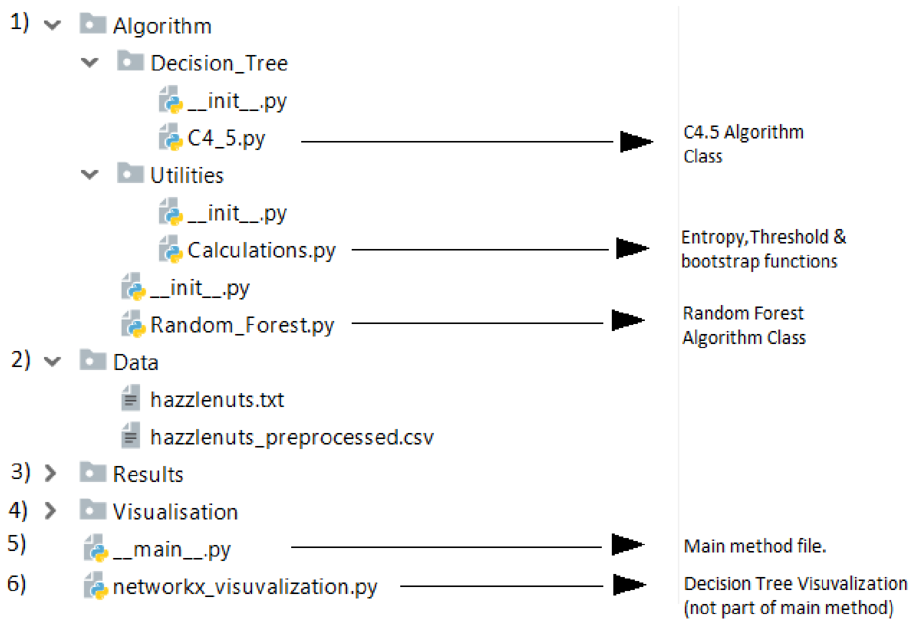
REFERENCES

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

```python
import random
import pandas as pd
import numpy as np
import pprint
from operator import itemgetter

# Files for this below 2 imports are attached

from Algorithm.Decision_Tree.C4_5 import * # CART implementation from scratch
from Algorithm.Utilities.Calculations import *   # All helper functions involving calculations




class random_forest():

    # At each tree we store features sent and their indexes which will be used for testing
    features_idx_subseted = []
    feature_subseted = []

    # We store Meta info in a dictionary with assumed structure { 'Tree_1' : 'Meta_Data_frame_of_tree1',.....}
    meta_dfs = {}
    def __init__ (self,data,total_trees,bootstrap_size,feature_per_tree,tree_depth = 5,random_seed = 10):

        self.feature_per_tree = feature_per_tree
        self.bootstrap_size = bootstrap_size
        self.total_trees = total_trees
        self.data = data
        self.features =data.columns.tolist()[0:-1]
        self.lable_name = data.columns.tolist()[-1]
        self.tree_depth = tree_depth
        self.random_seed = random_seed


    def fit(self):

        ''''Build each tree using Bootstrap sample Random Feature subspace.'''

        random.seed(self.random_seed)

        for tree in range(self.total_trees):
            sample = get_bootstrap(self.data,self.bootstrap_size,self.random_seed)

            sub_feature = random.sample(range(len(self.features)), k=self.feature_per_tree)
            random_forest.features_idx_subseted.append(sub_feature)

            feature_subset = list(itemgetter(*sub_feature)(self.features))
            random_forest.feature_subseted.append(feature_subset)

            # Call to decision tree

            c = DT_classifier(self.tree_depth,sample,feature_subset,self.lable_name,sample[self.lable_name].unique().tolist())

            c.fit()


            # Appending decision tree meta info of this tree to our random forest meta dictionary

            random_forest.meta_dfs[tree] = c.mdf


    def __predict_tree_op(self,ip_arr,cols,mdf,get_prob = 'N'):

        ''''Predicts lables for given tree for input array of attributes.

        Parameters:
            Attributes Array,Meta Data frame

        Summary:
            -Gets all Leaf Node conditions from Meta data frame
            -Checks which condition is met by test data and maps leaf node pertaining to the condition.
            -Predicts the most probable output class the probabilities in leaf node.

        Returns:
            Predicted lables for the tree passed'''

        op_lst = []
        leaf_node_df = mdf [mdf['leaf_f'] == 'Y']
        leaf_conditions = leaf_node_df['split_condition']
        for elm in ip_arr:
            p = pd.Series(elm,index = cols)
            for num,condition in enumerate(leaf_conditions):
                check = []
                for cond in condition:
                    col = cond.split()[0]
```

```
             val = ' '.join(cond.split()[1:])
             if  eval(str(p[col]) + val):
                 check.append('True')
             else:
                 check.append('False')
         if 'False' not in check:
             class_prob_df = leaf_node_df.iloc[num,][self.data[self.lable_name].unique().tolist()]
             if get_prob == 'Y':
                 op = class_prob_df.values.tolist()
                 op_lst.append(op)
             else:
                 op = class_prob_df[class_prob_df == class_prob_df.max()].index.values #returns column name with max value
                 op_lst.append(op[0])
             break

        return np.array(op_lst)


    def predict(self,ip_arr):

        '''''Predicts lables from output lables obtained from each tree.

        Parameters:
            Attributes Array,

        Returns:
            Predicted lables'''

        votes_arr = []

        for tree in range(self.total_trees):

            cols = random_forest.feature_subseted[tree]
            idx_cols = random_forest.features_idx_subseted[tree]

            idx_data =np.array([list(arr[idx_cols]) for arr in ip_arr])

            op = self.__predict_tree_op(idx_data,cols,random_forest.meta_dfs[tree])
            votes_arr.append(op)
            odf = pd.DataFrame(np.array(votes_arr))
        return odf.mode().iloc[0].values

    def write_meta_data(self,file_name):

        with pd.ExcelWriter(file_name) as workbook:
            for tree,mdf in random_forest.meta_dfs.items():
                mdf.to_excel(workbook, sheet_name='Tree_'+str(tree+1),index=False)
            print("\nCheck '{}' for meta information at each tree.".format(file_name))




    def predict_prob(self,ip_arr):

        '''''Predicts lables from output lables obtained from each tree.

        Parameters:
            Attributes Array,

        Returns:
            Predicted lables'''

        votes_arr = []

        for tree in range(self.total_trees):

            cols = random_forest.feature_subseted[tree]
            idx_cols = random_forest.features_idx_subseted[tree]

            idx_data =np.array([list(arr[idx_cols]) for arr in ip_arr])
```

```python

                    op = self.__predict_tree_op(idx_data,cols,random_forest.meta_dfs[tree],get_prob='Y')
                    votes_arr.append(op)
                prob = np.array(votes_arr)
                return np.mean(prob,axis=0)
```

C4_5.PY

```python
import pandas as pd
import numpy as np

from Algorithm.Utilities.Calculations import *


class DT_classifier():

    # Meta Data Frame to store decision tree information
    mdf  = pd.DataFrame()

    def __init__(self,max_depth,train_df,feature_list,lable_name,lable_list):
        self.max_depth = max_depth
        self.train_df = train_df
        self.feature_list = feature_list
        self.lable_name = lable_name
        self.lable_list = lable_list


    def __tree_initial_split(self,feature_list,lable,df_test):

        """Initiate initial tree split to get data into meta df for further spliting.

        Parameters -

            Attributes,Label name,Data

        Returns -

            Meta Data Frame
        """
        split_feature,threshold,overall_entropy = discretize(df_test,lable,feature_list)
        cols = ['node_number','parent_node','split_feature','split_category','split_condition','overall_entropy','leaf_f','next_split_features']
        val = [['1',np.nan,split_feature,np.nan,[threshold],overall_entropy,'N']]
        feature_for_next_split = feature_list.copy()
        feature_for_next_split.remove(split_feature)  # removing current split feature for further spliting
        val[0].append(feature_for_next_split)
        lables = df_test[lable].unique().tolist()
        for i in lables:
            cols.append(str(i))   # creating columns as per no of labels in op class
            class_data = len(df_test[df_test[lable] == i])
            total_data = len(df_test)
            val[0].append(class_data/total_data) # probability for that lable in the node
        meta_df = pd.DataFrame(val,columns=cols)
        #for num,levels in enumerate(df_test[split_feature].unique()): #Branches from the first node
        for num,operator in enumerate([' <= ',' > ']):
            condition = operator +str(threshold)
            vals = [['1_'+str(num+1),'1',split_feature,condition,[split_feature+condition],np.nan,np.nan,feature_for_next_split]]
            for lable in lables: vals[0].append(np.nan) #appending values for op class probability

            meta_df = meta_df.append(pd.DataFrame(vals,columns=cols),ignore_index=True)[cols] # adding observations to meta_df
        return meta_df





    def __build_tree(self,data,meta_df,lable_name,lable_list,final_split = 'N'):

        '''''Split tree from the provided state to next state.

        Parameters:

            Data,Meta Data Frame,Label name,Classes in lable,Final split flag

        Summary:

            -Filters the nodes that are not split using meta info
            -Filtered Nodes which are non-terminal will be split further.
            -Filtered Nodes are updated with Leaf flag,overall entropy and class probability information in meta data frame.
            -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)

        Returns:
            Temporary Data Frame containg meta information of new split done'''

        df_node_to_split = meta_df[(meta_df['parent_node'].notnull()) & (pd.isnull(meta_df['overall_entropy'])) ]
        cols = meta_df.columns.tolist()
        temp_df = pd.DataFrame(columns=cols)
        for i in range(0,len(df_node_to_split)):
            leaf = False
            series = df_node_to_split.iloc[i,]
            node_number,features_for_split,split_condition = series['node_number'],series['next_split_features'],series['split_condition']

            #Filtering data subset based on split condition dict
            subset = data
            for cond in split_condition:
                key = cond.split()[0]
                value = ' '.join(cond.split()[1:])
                subset = eval('subset[subset[key]'+ value+']')

            probs = []
            #Updating class label  propbailites and lead flag
            for lable in lable_list:
                if len(subset) >0: # for zreo division error
                    prob = len(subset[subset[lable_name] == lable]) / len(subset)
                    meta_df.loc[meta_df['node_number'] ==series['node_number'],str(lable)] = prob

                    probs.append(prob)
            if (1 in probs) or (final_split == 'Y') or (len(features_for_split) == 0):
                meta_df.loc[meta_df['node_number'] ==series['node_number'],'leaf_f'] = 'Y'
                meta_df.loc[meta_df['node_number'] ==series['node_number'],'overall_entropy'] = 0

                leaf = True
            else:
                meta_df.loc[meta_df['node_number'] ==series['node_number'],'leaf_f'] = 'N'


            if leaf or (final_split == 'Y') :
                continue

            #Get split feature for further split if its not a leaf node

            split_feature,threshold,overall_entropy = discretize(subset,lable_name,features_for_split)
            meta_df.loc[meta_df['node_number'] ==series['node_number'],'overall_entropy'] = overall_entropy #Weighted entropy update
            next_features = features_for_split.copy()
            next_features.remove(split_feature) # child's next feature for split

            #child observations entry
            for num,operator in enumerate([' <= ',' > ']):
```

```
            #for num,levels in enumerate(subset[split_feature].unique()):
                condition = split_condition.copy()
                condition.append(split_feature + operator + str(threshold))
                vals = [[node_number+'_'+str(num+1),node_number,split_feature,operator + str(threshold),condition,np.nan,np.nan,next_features]]
                for lbl in lable_list: vals[0].append(np.nan)
                temp_df = temp_df.append(pd.DataFrame(vals,columns=cols),ignore_index=True)[cols]
        return temp_df

    def __recursive_tree(self,data,max_depth,lable_name,feature_list,lable_list):

        '''''Initiates tree building and splits child nodes until convergence.

        Parameters:
            Data,Tree Depth,Label name,Data,Attributes,Output class names

        Returns:
            Meta Data Frame'''

        meta_df = self.__tree_initial_split (feature_list,lable_name,data)
        max_depth = max_depth - 1 #as already 1 split has been done by us
        for i in range(0,max_depth):
            temp_df = self.__build_tree(data,meta_df,lable_name,lable_list)
            meta_df = meta_df.append(temp_df,ignore_index=True)[meta_df.columns.tolist()]
        temp_df = self.__build_tree(data,meta_df,lable_name,lable_list,final_split = 'Y') # just to update meta data no split occurs here
        meta_df = meta_df.append(temp_df,ignore_index=True)[meta_df.columns.tolist()]
        return meta_df




    def fit(self):
        df = self.__recursive_tree(self.train_df,self.max_depth,self.lable_name,self.feature_list,self.lable_list)
        DT_classifier.mdf = df
```

CALCULATIONS.PY

```
import numpy as np

def get_threshold(df,col,lable):

    '''''Get best threshold value for the attribute

        Parameters :
            Data frame,Attribute,Lable name

        Returns:
            Entropy value,Threshold value'''

    df = df[[col,lable]]
    uniq_vals = df[col].unique().tolist()
    uniq_vals.sort()
    thresholds = [(uniq_vals[idx]+uniq_vals[idx+1])/2 for idx in range(0,len(uniq_vals) -1)]
    weighted_entropy = []

    for threshold in thresholds:
        left = df[df[col] <= threshold][lable]
        right = df[df[col] > threshold][lable]
        entropy = calc_wgtd_entropy_numeric(left,right)
        weighted_entropy.append(entropy)
    if thresholds == []:
        weighted_entropy = [0]
        thresholds = [0]
```

```python

    return [min(weighted_entropy),thresholds[weighted_entropy.index(min(weighted_entropy))]]



def calc_wgtd_entropy_numeric( left, right):

    '''''Get overall entropy for an attribute

        Parameters :
            Data towards left and right of threshold (i.e, <= threshold & > threshold )

        Returns:
            Overall entropy'''

    total_elements = len(left) + len(right)
    ent_left = entropy(left)
    ent_right = entropy(right)
    weighted_entropy = ((len(left) / total_elements) * ent_left) + ((len(right) / total_elements) * ent_right)
    return weighted_entropy


def entropy(df):

    '''''Get entropy for a subset

        Parameters :
            Lable values of subset

        Returns:
            Entropy'''

    op_class, count = np.unique(df.values, return_counts=True)
    entropy = np.sum([(-count[i] / np.sum(count)) * np.log2(count[i] / np.sum(count)) for i in range(len(op_class))])
    return entropy



def discretize(df,lable,feature_list):

    '''''Discretization (binning) of continuous data

        Parameters :
            Data Frame,Lable Name,Attributes

        Returns:
            Attribute,Threshold,Overall Entropy'''

    val = []
    d = {}
    for i in feature_list:
        op = get_threshold(df,i,lable)
        val.append(op)
        d[op[1]] = i

    ent = [value[0] for value in val]
    thr = [value[1] for value in val]
    return d[thr[ent.index(min(ent))]],thr[ent.index(min(ent))],min(ent)


def get_bootstrap(data, bootstrap_size,random_seed):

    ''''' Bootstraping for random forrest

        Parameters :
            Data Frame, Bootstarp size, Random state

        Returns:
            Boostrap sample'''

    np.random.seed(random_seed)
    bootstraps = np.random.randint(low=0, high=len(data), size=bootstrap_size)
    df_bootstrap = data.iloc[bootstraps]
            return df_bootstrap
```

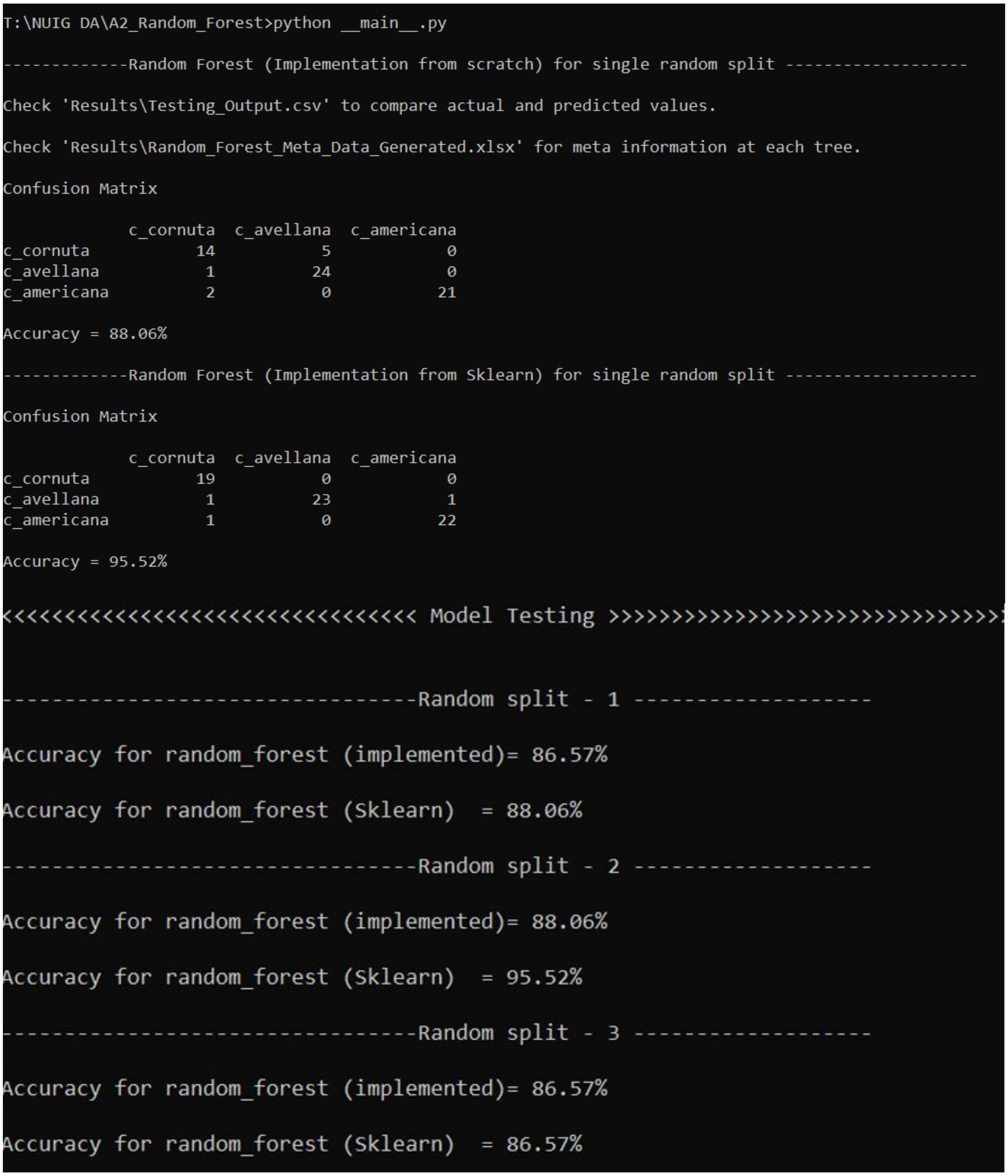
__MAIN__.PY

```python
import random
import pandas as pd
import numpy as np
from Algorithm.Random_Forest import random_forest
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.model_selection import cross_val_score,cross_val_predict

df = pd.read_csv('Data\\hazzlenuts_preprocessed.csv')

features = df.columns[0:-1].tolist()
output_lable_name = df.columns[-1]


x = df[features].values
y = df[output_lable_name].values

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.33, random_state = 1)

train_df = pd.DataFrame(x_train,columns = features)
train_df[output_lable_name] = y_train
train_df

def gen_random_splits(n):
    sk_rf_acc = []
    rf_acc = []
    for split in range(0,n):

        print('\n----------------------------------Random split - {} -------------------
'.format(split+1))
        x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.33, random_state =
 split)

        train_df = pd.DataFrame(x_train,columns = features)
        train_df[output_lable_name] = y_train
        train_df

        r = random_forest(data = train_df,total_trees = 11,tree_depth = 5,random_seed = 10,feature
_per_tree = 6,bootstrap_size=100)
        r.fit()
        op= r.predict(x_test)
        acc_rf = metrics.accuracy_score(y_test, op)
        print('\nAccuracy for random_forest (implemented)= {}%'.format(round(acc_rf*100,2)))
        rf_acc.append(acc_rf)

        classifier = RandomForestClassifier(n_estimators = 11,max_features = 6, criterion = 'entro
py', random_state = 10,max_depth = 5)
        classifier.fit(x_train, y_train)
        op = classifier.predict(x_test)
        acc_sk = metrics.accuracy_score(y_test, op)
        print('\nAccuracy for random_forest (Sklearn)  = {}%'.format(round(acc_sk*100,2)))
        sk_rf_acc.append(acc_sk)

    print('\n-------------------------------------------------------')
    print('\nMean Accuracy for random_forest (Implemented) after {} random splits  = {}%'.format(n
,round(np.array(rf_acc).mean()*100,3)))
```

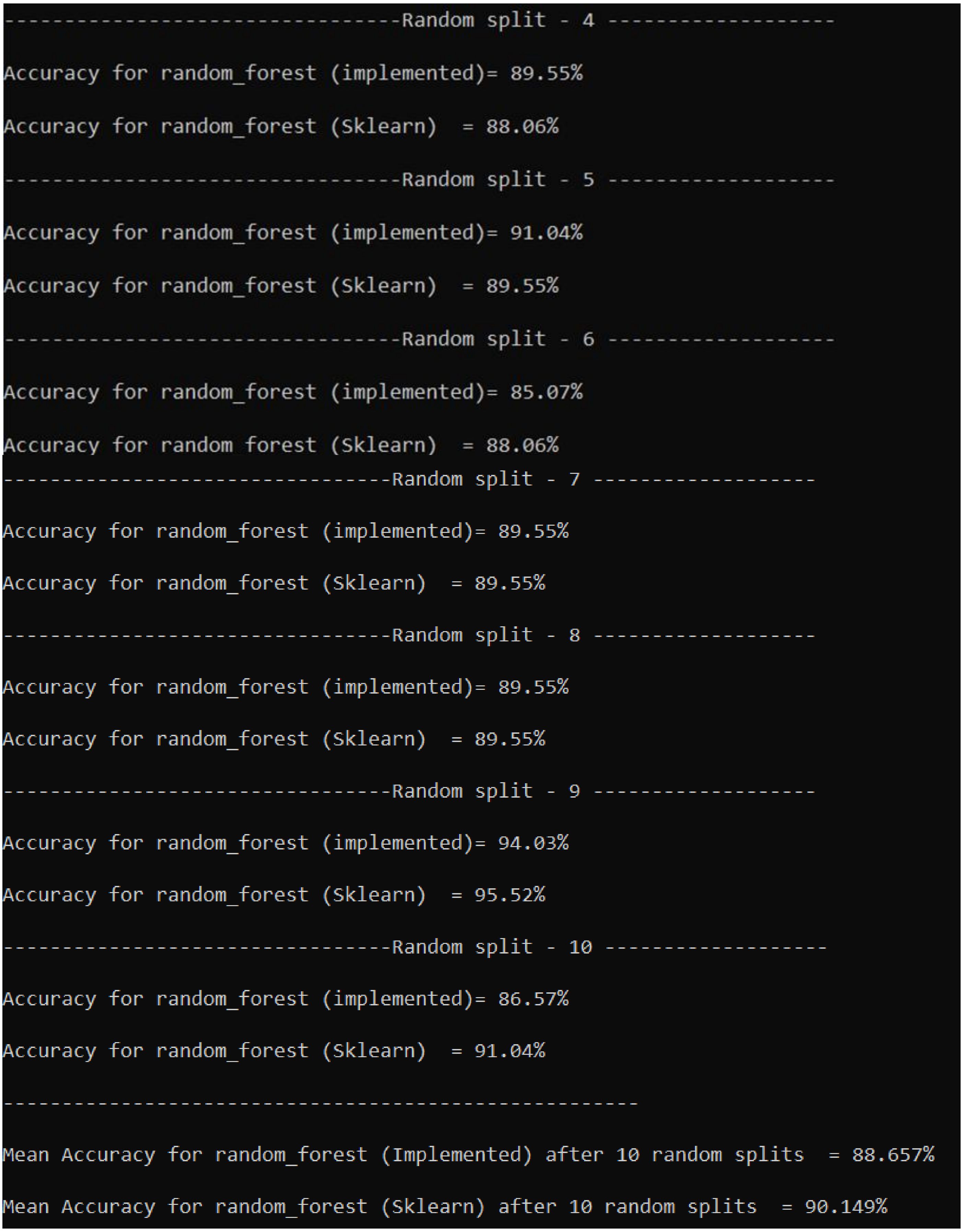
```python
    print('\nMean Accuracy for random_forest (Sklearn) after {} random splits  = {}%'.format(n,round(np.array(sk_rf_acc).mean()*100,3)))

if __name__ == "__main__":

    print('\n-------------Random Forest (Implementation from scratch) for single random split -------------------')
    r = random_forest(data = train_df,total_trees = 11,tree_depth = 5,random_seed = 10,feature_per_tree = 6,bootstrap_size=100)

    r.fit()
    op= r.predict(x_test)
    prob_arr = r.predict_prob(x_test)
    pd.DataFrame({'Actual': y_test,'Predicted':op}).to_csv('Results\\Testing_Output.csv',index=False)

    print("\nCheck 'Results\\Testing_Output.csv' to compare actual and predicted values.")
    r.write_meta_data('Results\\Random_Forest_Meta_Data_Generated.xlsx')

    print('\nConfusion Matrix \n')
    cm = confusion_matrix(y_test, op)
    print(pd.DataFrame(cm,columns=train_df.iloc[:,-1].unique(),index = train_df.iloc[:,-1].unique() ))


    acc = metrics.accuracy_score(y_test, op)
    print('\nAccuracy = {}%\n'.format(round(acc*100,2)))

    print('-------------Random Forest (Implementation from Sklearn) for single random split --------------------')


    classifier = RandomForestClassifier(n_estimators = 11,max_features = 6, criterion = 'entropy', random_state = 10,max_depth = 5)
    classifier.fit(x_train, y_train)
    op = classifier.predict(x_test)

    print('\nConfusion Matrix\n')
    cm_sk = confusion_matrix(y_test, op)
    print(pd.DataFrame(cm_sk,columns=train_df.iloc[:,-1].unique(),index = train_df.iloc[:,-1].unique() ))

    acc_sk = metrics.accuracy_score(y_test, op)
    print('\nAccuracy = {}%'.format(round(acc_sk*100,2)))


    print('\n\n<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Model Testing >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n')
    gen_random_splits(10)

```

CONSOLE OUTPUT:

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



```
T:\NUIG DA\A2_Random_Forest>python __main__.py

------------Random Forest (Implementation from scratch) for single random split ------------------

Check 'Results\Testing_Output.csv' to compare actual and predicted values.

Check 'Results\Random_Forest_Meta_Data_Generated.xlsx' for meta information at each tree.

Confusion Matrix

             c_cornuta  c_avellana  c_americana
c_cornuta           14           5            0
c_avellana           1          24            0
c_americana          2           0           21

Accuracy = 88.06%

------------Random Forest (Implementation from Sklearn) for single random split -------------------

Confusion Matrix

             c_cornuta  c_avellana  c_americana
c_cornuta           19           0            0
c_avellana           1          23            1
c_americana          1           0           22

Accuracy = 95.52%

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Model Testing >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>


--------------------------------Random split - 1 ------------------

Accuracy for random_forest (implemented)= 86.57%

Accuracy for random_forest (Sklearn)  = 88.06%

--------------------------------Random split - 2 ------------------

Accuracy for random_forest (implemented)= 88.06%

Accuracy for random_forest (Sklearn)  = 95.52%

--------------------------------Random split - 3 ------------------

Accuracy for random_forest (implemented)= 86.57%

Accuracy for random_forest (Sklearn)  = 86.57%
```



```
---------------------------------Random split - 4 -------------------

Accuracy for random_forest (implemented)= 89.55%

Accuracy for random_forest (Sklearn)  = 88.06%

---------------------------------Random split - 5 -------------------

Accuracy for random_forest (implemented)= 91.04%

Accuracy for random_forest (Sklearn)  = 89.55%

---------------------------------Random split - 6 -------------------

Accuracy for random_forest (implemented)= 85.07%

Accuracy for random forest (Sklearn)  = 88.06%
---------------------------------Random split - 7 -------------------

Accuracy for random_forest (implemented)= 89.55%

Accuracy for random_forest (Sklearn)  = 89.55%

---------------------------------Random split - 8 -------------------

Accuracy for random_forest (implemented)= 89.55%

Accuracy for random_forest (Sklearn)  = 89.55%

---------------------------------Random split - 9 -------------------

Accuracy for random_forest (implemented)= 94.03%

Accuracy for random_forest (Sklearn)  = 95.52%

---------------------------------Random split - 10 -------------------

Accuracy for random_forest (implemented)= 86.57%

Accuracy for random_forest (Sklearn)  = 91.04%

------------------------------------------------------

Mean Accuracy for random_forest (Implemented) after 10 random splits  = 88.657%

Mean Accuracy for random_forest (Sklearn) after 10 random splits  = 90.149%
```