Data Mining and Web algorithm

Lab Assignment 6:

[11 – 16 Apr, 2022]

Patil Amit Gurusidhappa
19104004
B11

Q1: Consider the dataset having four features: Branch, CGPA, Gamer and Movie Fanatic and the
class Committed.
a) Compute the Information Gain of the features that better discriminates the class committed.
b) Then split the tree based on the first feature. Choose the further features based on the information gains.
c) Print the information gains for each feature and every split.
d) Finally print the tree.
e) Test the model with the below samples and calculate the accuracy.
f) Implement the algorithm in python and weka tool.
g) Perform the computation on your notebook and compare the results obtained in part f.

```
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
```

```
df=pd.read_csv('E:/Work/JIIT/sem_6/JIIT-SEM-6/DataMining&WebAlgorithms/Lab 6/assgn6_q1_dataset.csv');
print(df)
```

	S no.	branch	cgpa	gamer	movie_fanatic	commited
0	1	CSE	high	yes	no	no
1	2	CSE	low	yes	no	no
2	3	CSE	high	yes	yes	no
3	4	CSE	high	no	no	yes
4	5	CSE	low	no	yes	yes
5	6	ECE	low	yes	no	no
6	7	ECE	high	yes	yes	yes
7	8	ECE	low	yes	yes	no
8	9	ECE	high	yes	yes	yes
9	10	ECE	high	no	yes	yes
10	11	MECH	high	yes	yes	no
11	12	MECH	high	no	no	no
12	13	MECH	high	no	no	yes
13	14	MECH	low	no	no	yes
14	15	MECH	low	no	no	yes
15	16	CSE	high	yes	no	no
16	17	ECE	low	yes	no	no

```
inputs=df.drop('commited',axis='columns')
inputs
```

	S no.	branch	cgpa	gamer	movie_fanatic
0	1	CSE	high	yes	no
1	2	CSE	low	yes	no
2	3	CSE	high	yes	yes
3	4	CSE	high	no	no
4	5	CSE	low	no	yes
5	6	ECE	low	yes	no
6	7	ECE	high	yes	yes
7	8	ECE	low	yes	yes
8	9	ECE	high	yes	yes
9	10	ECE	high	no	yes
10	11	MECH	high	yes	yes
11	12	MECH	high	no	no

```
from sklearn.preprocessing import LabelEncoder
```

```
le_branch=LabelEncoder()
le_cgpa =LabelEncoder()
le_gamer =LabelEncoder()
le_movie_fanatic =LabelEncoder()
le_commited=LabelEncoder()
```

```
# creating new columns
inputs['branch_n']=le_branch.fit_transform(inputs['branch'])
inputs['cgpa_n']=le_cgpa.fit_transform(inputs['cgpa'])
inputs['gamer_n']=le_gamer.fit_transform(inputs['gamer'])
```

```
inputs['movie_fanatic_n']=le_movie_fanatic.fit_transform(inputs['movie_fan
atic'])
df['commited_n']=le_commited.fit_transform(df['commited'])
```

```
inputs
```

Python

	S no.	branch	cgpa	gamer	movie_fanatic	branch_n	cgpa_n	game
0	1	CSE	high	yes	no	0	0	
1	2	CSE	low	yes	no	0	1	
2	3	CSE	high	yes	yes	0	0	
3	4	CSE	high	no	no	0	0	
4	5	CSE	low	no	yes	0	1	
5	6	ECE	low	yes	no	1	1	
6	7	ECE	high	yes	yes	1	0	
7	8	ECE	low	yes	yes	1	1	
8	9	ECE	high	yes	yes	1	0	
9	10	ECE	high	no	yes	1	0	
10	11	MECH	high	yes	yes	2	0	

```
# dropping non relavent data columns
inputs_n=inputs.drop(['S
no.','branch','cgpa','gamer','movie_fanatic'],axis='columns');
```

```
from sklearn import tree
```

```
model=tree.DecisionTreeClassifier()
```

```
model.fit(inputs_n,target)
```

```
model.score(inputs_n,target)
```

```
0.9444444444444444
```

Q2: Download 1 classification dataset (https://tinyurl.com/uciclass)
a) Load the data, pre-process the data.
b) Built a decision tree using ID3 Algorithm in python.

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
```

```
col_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'type']
data = pd.read_csv("iris.csv", skiprows=1, header=None, names=col_names)
data.head(10)
```

	sepal_length	sepal_width	petal_length	petal_width	type
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
2	4.7	3.2	1.3	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
4	5.0	3.6	1.4	0.2	Setosa
5	5.4	3.9	1.7	0.4	Setosa
6	4.6	3.4	1.4	0.3	Setosa
7	5.0	3.4	1.5	0.2	Setosa
8	4.4	2.9	1.4	0.2	Setosa
9	4.9	3.1	1.5	0.1	Setosa

```
lebal_type=LabelEncoder()
data['type_n']=lebal_type.fit_transform(data['type'])
data1=data.drop('type',axis='columns')
data1.head(n=600)
```

	sepal_length	sepal_width	petal_length	petal_width	type_n
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2

Node Class

```
class Node():
    def __init__(self, feature_index=None, threshold=None, left=None,
right=None, info_gain=None, value=None):
        ''' constructor '''

        # for decision node
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain

        # for leaf node
        self.value = value
```

```
class DecisionTreeClassifier():
```

```
    def __init__(self, min_samples_split=2, max_depth=2):
        ''' constructor '''

        # initialize the root of the tree
        self.root = None

        # stopping conditions
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth

    def build_tree(self, dataset, curr_depth=0):
        ''' recursive function to build the tree '''

        X, Y = dataset[:,:-1], dataset[:,-1]
        num_samples, num_features = np.shape(X)


        # split until stopping conditions are met
        if num_samples>=self.min_samples_split and
curr_depth<=self.max_depth:
            # find the best split
            best_split = self.get_best_split(dataset, num_samples,
num_features)
            # check if information gain is positive
            if best_split["info_gain"]>0:
                # recur left
                left_subtree = self.build_tree(best_split["dataset_left"],
curr_depth+1)
                # recur right
                right_subtree =
self.build_tree(best_split["dataset_right"], curr_depth+1)
                # return decision node
                return Node(best_split["feature_index"],
best_split["threshold"],
                            left_subtree, right_subtree,
best_split["info_gain"])

        # compute leaf node
        leaf_value = self.calculate_leaf_value(Y)
        # return leaf node
```

```
        return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
        ''' function to find the best split '''

        # dictionary to store the best split
        best_split = {}
        max_info_gain = -float("inf")

        # loop over all the features
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            possible_thresholds = np.unique(feature_values)
            # loop over all the feature values present in the data
            for threshold in possible_thresholds:
                # get current split
                dataset_left, dataset_right = self.split(dataset,
feature_index, threshold)
                # check if childs are not null
                if len(dataset_left)>0 and len(dataset_right)>0:
                    y, left_y, right_y = dataset[:, -1], dataset_left[:,
-1], dataset_right[:, -1]
                    # compute information gain
                    curr_info_gain = self.information_gain(y, left_y,
right_y, "gini")
                    # update the best split if needed
                    if curr_info_gain>max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain

        # return best split
        return best_split

    def split(self, dataset, feature_index, threshold):
        ''' function to split the data '''
```

```
        dataset_left = np.array([row for row in dataset if
row[feature_index]<=threshold])
        dataset_right = np.array([row for row in dataset if
row[feature_index]>threshold])
        return dataset_left, dataset_right

    def information_gain(self, parent, l_child, r_child, mode="entropy"):
        ''' function to compute information gain '''

        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
        if mode=="gini":
            gain = self.gini_index(parent) -
(weight_l*self.gini_index(l_child) + weight_r*self.gini_index(r_child))
        else:
            gain = self.entropy(parent) - (weight_l*self.entropy(l_child)
+ weight_r*self.entropy(r_child))
        return gain

    def entropy(self, y):
        ''' function to compute entropy '''

        class_labels = np.unique(y)
        entropy = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            entropy += -p_cls * np.log2(p_cls)
        return entropy

    def gini_index(self, y):
        ''' function to compute gini index '''

        class_labels = np.unique(y)
        gini = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            gini += p_cls**2
        return 1 - gini

    def calculate_leaf_value(self, Y):
```

```
        ''' function to compute leaf node '''

        Y = list(Y)
        return max(Y, key=Y.count)

    def print_tree(self, tree=None, indent=" "):
        ''' function to print the tree '''

        if not tree:
            tree = self.root

        if tree.value is not None:
            print(tree.value)

        else:
            print("X_"+str(tree.feature_index), "<=", tree.threshold, "?",
tree.info_gain)
            print("%sleft:" % (indent), end="")
            self.print_tree(tree.left, indent + indent)
            print("%sright:" % (indent), end="")
            self.print_tree(tree.right, indent + indent)

    def fit(self, X, Y):
        ''' function to train the tree '''

        dataset = np.concatenate((X, Y), axis=1)
        self.root = self.build_tree(dataset)

    def predict(self, X):
        ''' function to predict new dataset '''

        preditions = [self.make_prediction(x, self.root) for x in X]
        return preditions

    def make_prediction(self, x, tree):
        ''' function to predict a single data point '''

        if tree.value!=None: return tree.value
        # print(tree.feature_index)
        feature_val = x[tree.feature_index]
```

```
        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)
```

Train-Test split

```
X = data1.iloc[:, :-1].values
Y = data1.iloc[:, -1].values.reshape(-1,1)
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.2,
random_state=41)
```

Fit the model

```
classifier = DecisionTreeClassifier(min_samples_split=3, max_depth=3)
classifier.fit(X_train,Y_train)
classifier.print_tree()
```

Decision Tree

```
[52]

...   X_2 <= 1.9 ? 0.337413853727144 94
       left:0.0
       right:X_3 <= 1.5 ? 0.427106638180289
        left:X_2 <= 4.9 ? 0.051246537396121 73
          left:1.0
          right:2.0
        right:X_2 <= 5.0 ? 0.019631171921475288
          left:X_1 <= 2.8 ? 0.20833333333333334
              left:2.0
              right:1.0
          right:2.0
```