INF 552 MACHINE LEARNING FOR DATA SCIENCE

HOMEWORK 1

Team member:

SANJAY MALLASAMUDRAM SANTHANAM – 3124715393

Part 1:

Language used:

Python

Data structures used:

Pandas Dataframe: Pandas dataframe was used to read read and manipulate training data. No other special data structure is used.

Dictionaries and trees were used to represent decision tree.

Code-level optimization:

Save execution time by selecting best attribute based on lowest entropy of split attribute instead of calculating information gain because we know Info gain=entropy(parent_data)-entropy(split_data). Entropy (parent_data) is the same while calculating Info gain of all attributes. So what we essentially do is subtract it from a constant value and find max Info gain. Instead logically, we can shoose lowest entropy among all attributes,which will give the same best attribute.Since our aim is to find the best attribute to split, we don't care about is numerical value. Hence this method can be used to optimize decision tree splitting.

Challenges:/ DECISION TREE FORMAT

Representing decision tree in a non-graphical format required some thinking. Decision tree is stored as a combination of list and dictionary, with the dictionary key being attribute name (or) attribute value and dictionary value being a list(array) of collection of all possible values the attribute can take, if the key is attribute name (or) next attribute to split if key is attribute value . If a dictionary key is attribute name, the dictionary value is a list consisting of dictionaries of the attribute values. If the dictionary's key is an attribute, the dictionary value is either a label(denoting leaf node) or next attributes which is used to split the tree.

Also, if there is a tie between two variables to split, the one that occurs first in the table is taken.

Prediction:

Yes

PART 2:

Library used: Scikit-Learn.

Scikit learn has a function DecisionTreeClassifier() that fits a Decision tree for given data and label. The algorithm is not exactly greedy like ID3, but a combination of many algorithms. The decision tree constructed by this method is better than ID3 because it trains multiple decision tree in an ensemble learner, with the features and samples selected randomly with replacement. Hence the output of this method is a smaller and compact decision tree compared to ID3. To improve on ID3, We can use bagging method which is similar to what the library method does.

Part 3:

Decision trees are used in Fraudulent Financial statement detection. Decision trees are proved to be giving better accuracy than standard statistical models. Features used are financial statements like income statement, balance sheets etc.

Decision trees are also used to detect defects in machinaries. The vibrations and the acoustic emissions from the machines are used as features. But to measure these factors, a lot of irrelevant varialbes are involved which are eliminated by decision trees.

In [1]:
```python
#Done by SANJAY MALLASAMUDRAM SANTHANAM ; USC ID:3124715393
import pandas as pd
import numpy as np
import math
```

In [2]:
```python
#read data
data=pd.read_csv('C:/Users/Lenovo/Desktop/data.txt', sep=", ", header=None,names=['Occupied'
, 'Price', 'Music', 'Location', 'VIP', 'Favorite Beer', 'Enjoy'])
```

```
C:\Users\Lenovo\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: ParserWarning: Falling
back to the 'python' engine because the 'c' engine does not support regex separators (separa
tors > 1 char and different from '\s+' are interpreted as regex); you can avoid this warning
by specifying engine='python'.
```

In [3]:
```python
data.head()
```

Out[3]:

	Occupied	Price	Music	Location	VIP	Favorite Beer	Enjoy
0	(Occupied	Price	Music	Location	VIP	Favorite Beer	Enjoy)
1	01: High	Expensive	Loud	Talpiot	No	No	No;
2	02: High	Expensive	Loud	City-Center	Yes	No	Yes;
3	03: Moderate	Normal	Quiet	City-Center	No	Yes	Yes;
4	04: Moderate	Expensive	Quiet	German-Colony	No	No	No;

In [4]:
```python
#since file is .txt , remove 1st row. Headers defined manually.
data=data.drop(data.index[0])
```

In [5]:
```python
#remove row number from 1st attribute and semicolon from last attribute. Since separator was
comma, these have to be manually removed.
for i in range(0,len(data)):
    data["Occupied"].values[i]=data["Occupied"].values[i].split(":")[1]
    data["Enjoy"].values[i]=data["Enjoy"].values[i].split(";")[0]
#remove white speaces from all attributes.
for i in data.columns:
    data[i]=data[i].str.strip()
```

In [6]:
```python
#converted the data to required format after applying necessary pre-processing
data.head()
```

Out[6]:

	Occupied	Price	Music	Location	VIP	Favorite Beer	Enjoy
1	High	Expensive	Loud	Talpiot	No	No	No
2	High	Expensive	Loud	City-Center	Yes	No	Yes
3	Moderate	Normal	Quiet	City-Center	No	Yes	Yes
4	Moderate	Expensive	Quiet	German-Colony	No	No	No
5	Moderate	Expensive	Quiet	German-Colony	Yes	Yes	Yes

In [7]:
```
#function to calculat entropy.
def ent(data):
    #count diff value of attributes in final attribute i.e. label attribute.
    label_count=data[data.columns[-1]].value_counts(normalize=True)
    en=0
    for i in label_count:
        #to prevent math error(domain error) when performing log operation, if the value is
 0, we skip it as adding 0 doesnt make any differene
        if(i==0):
            continue
        en+=-1*i*math.log2(i)
    return en
```

In [8]:
```
#given data, returns the best attribute to split data. ie. returns attribute with lowest ent
ropy.
#calculating information gain is redundant once we find entropy because we just subtract ent
ropy value form a constant value.
#instead of subtracting entropy and choosing the highest Information gain, we can choose low
est entropy without subtracting as both methods give same result.

def best(data):
    cols=data.columns
    #choose all columns except last label attribute.
    cols=cols[:-1]
    best_col=''
    #entropy value cant exceed 1 so initial value set as 2
    min_val=2
    for col in cols:
        e=0
        #find the fractional count of occurrence of each attribute.
        s=data[col].value_counts(normalize=True)
        for v in s.index:
            #choose row with specific value.
            temp=data[data[col]==v]
            e+=(s[v]*ent(temp))
        #if current entripy value is lower than minimum entropy value so far calculated
        if(e<min_val):
            min_val=e
            best_col=col
        #print(col,e)
    #print(data)
    return best_col
```

In [9]:
```
#this function selects the most common output value among a set of examples
def plur(data):
    #return label with maximum occurence
    return data[data.columns[-1]].max()
```

In [10]:
```
def dec_tree(data,attr,parent_data):
    #if all attributes are explored i.e. only label column is left, return most common label value
    if(len(attr)==1):
        return plur(data)
    #if all data rows are explored, return most common label of parent_data i.e. data before splitting attribute.
    elif(len(data)==0):
        return plur(parent_data)
    #if labels of data are the same, return the label i.e. check if it is a leaf node.
    elif(len(data[data.columns[-1]].value_counts())==1):
        return data[data.columns[-1]].values[0]
    #choose best attribute to split.
    a=best(data)
    #store decision tree in dictionary format with dictionary key being attribute and dictionary value being a list(array) of all
    #values of the best attribute. The value is further recursively stored as a dictionary with dictionary key being attribute value and
    #dictioniary value being next attribute that is split. i.e. recursively store decision tree as dictionary of values
    root={a:[]}
    V=data[a].value_counts(normalize=True)
    #print("Attr:",a,"Val:",V)
    for v in V.index:
        #filter data with specific value of attribute
        data_v=data[data[a]==v]
        #delete split attribute as we have already done using it nd dont need it for further splitting.
        del data_v[a]
        #recursively add child node of the decision tree to the parent node. Here attribute value is the dictionary key and next best
        #attribute to split is the dictionary value.For one level attribute name is key and for next level attribute value is key and so on.
        root[a].append({v:[dec_tree(data_v,attr.drop(a),data)]})
    #print(root)
    #return subree
    return root
```

In [11]:
```
#print decision tree
dt=dec_tree(data,data.columns,data)
```

In [12]:
```
dt
```

Out[12]:
```
{'Occupied': [{'Moderate': [{'Location': [{'Mahane-Yehuda': ['Yes']},
      {'German-Colony': [{'VIP': [{'No': ['No']}, {'Yes': ['Yes']}]}]},
      {'Ein-Karem': ['Yes']},
      {'Talpiot': [{'Price': [{'Normal': ['Yes']}, {'Cheap': ['No']}]}]},
      {'City-Center': ['Yes']}]}]},
  {'Low': [{'Location': [{'City-Center': [{'Price': [{'Normal': [{'Music': [{'Quiet': [{'VIP': [{'No': [{'Favorite Beer': [{'No': ['Yes']}]}]}]}]}]}]},
          {'Cheap': ['No']}]}]},
      {'Ein-Karem': [{'Price': [{'Cheap': ['Yes']}, {'Normal': ['No']}]}]},
      {'Talpiot': ['No']},
      {'Mahane-Yehuda': ['No']}]}]},
  {'High': [{'Location': [{'City-Center': ['Yes']},
      {'Mahane-Yehuda': ['Yes']},
      {'German-Colony': ['No']},
      {'Talpiot': ['No']}]}]}]}
```

In [13]:

```python
#test data
test={'Occupied':'Moderate','Price':'Cheap','Music':'Loud','Location':'City-Center','VIP':'No','Favorite Beer':'No'}
temp=dt
#travel the decision tree and when leaf is reached break the loop. Leaf is stored as a list and not as dictionary.
while(type(temp)==dict):
    for vv in temp.values():
        for v in vv:
            if(list(v.keys())[0]==test[list(temp.keys())[0]]):
                temp=list(v.values())[0][0]
                break
print("Predicted answer:",temp)
```

```
Predicted answer: Yes
```

In [ ]:

In [14]:

```python
dt
```

Out[14]:

```
{'Occupied': [{'Moderate': [{'Location': [{'Mahane-Yehuda': ['Yes']},
      {'German-Colony': [{'VIP': [{'No': ['No']}, {'Yes': ['Yes']}]}]},
      {'Ein-Karem': ['Yes']},
      {'Talpiot': [{'Price': [{'Normal': ['Yes']}, {'Cheap': ['No']}]}]},
      {'City-Center': ['Yes']}]}]},
  {'Low': [{'Location': [{'City-Center': [{'Price': [{'Normal': [{'Music': [{'Quiet': [{'VIP': [{'No': [{'Favorite Beer': [{'No': ['Yes']}]}]}]}]}]}]}]},
          {'Cheap': ['No']}]}]},
      {'Ein-Karem': [{'Price': [{'Cheap': ['Yes']}, {'Normal': ['No']}]}]},
      {'Talpiot': ['No']},
      {'Mahane-Yehuda': ['No']}]}]},
  {'High': [{'Location': [{'City-Center': ['Yes']},
      {'Mahane-Yehuda': ['Yes']},
      {'German-Colony': ['No']},
      {'Talpiot': ['No']}]}]}]}
```

In [15]:
```
data
```

Out[15]:

	Occupied	Price	Music	Location	VIP	Favorite Beer	Enjoy
1	High	Expensive	Loud	Talpiot	No	No	No
2	High	Expensive	Loud	City-Center	Yes	No	Yes
3	Moderate	Normal	Quiet	City-Center	No	Yes	Yes
4	Moderate	Expensive	Quiet	German-Colony	No	No	No
5	Moderate	Expensive	Quiet	German-Colony	Yes	Yes	Yes
6	Moderate	Normal	Quiet	Ein-Karem	No	No	Yes
7	Low	Normal	Quiet	Ein-Karem	No	No	No
8	Moderate	Cheap	Loud	Mahane-Yehuda	No	No	Yes
9	High	Expensive	Loud	City-Center	Yes	Yes	Yes
10	Low	Cheap	Quiet	City-Center	No	No	No
11	Moderate	Cheap	Loud	Talpiot	No	Yes	No
12	Low	Cheap	Quiet	Talpiot	Yes	Yes	No
13	Moderate	Expensive	Quiet	Mahane-Yehuda	No	Yes	Yes
14	High	Normal	Loud	Mahane-Yehuda	Yes	Yes	Yes
15	Moderate	Normal	Loud	Ein-Karem	No	Yes	Yes
16	High	Normal	Quiet	German-Colony	No	No	No
17	High	Cheap	Loud	City-Center	No	Yes	Yes
18	Low	Normal	Quiet	City-Center	No	No	No
19	Low	Expensive	Loud	Mahane-Yehuda	No	No	No
20	Moderate	Normal	Quiet	Talpiot	No	No	Yes
21	Low	Normal	Quiet	City-Center	No	No	Yes
22	Low	Cheap	Loud	Ein-Karem	Yes	Yes	Yes

In [16]:
```
from sklearn import tree
clf = tree.DecisionTreeClassifier(criterion='entropy')
data1=data
#test data for which prediction is needed
testing=pd.DataFrame({'Occupied':['Moderate'],'Price':['Cheap'],'Music':['Loud'],'Location':
['City-Center'],'VIP':['No'],'Favorite Beer':['No']})
```

In [17]:
```
from sklearn.preprocessing import LabelEncoder
#The library method does not deal with labelled data, instead it onyl works with numerical d
ata. So convert each attribute to
#a unique number. This is what LabelEncoder() function does.

for c in data1.columns:
    enc1=LabelEncoder()
    #since test data doesnt have label, we omit encoding it when we are encoding label attri
bute
    if(c!='Enjoy'):
        testing[c]=enc1.fit_transform(testing[c])
    data1[c]=enc1.fit_transform(data1[c])
```

In [18]:
```
data1
```

Out[18]:

	Occupied	Price	Music	Location	VIP	Favorite Beer	Enjoy
1	0	1	0	4	0	0	0
2	0	1	0	0	1	0	1
3	2	2	1	0	0	1	1
4	2	1	1	2	0	0	0
5	2	1	1	2	1	1	1
6	2	2	1	1	0	0	1
7	1	2	1	1	0	0	0
8	2	0	0	3	0	0	1
9	0	1	0	0	1	1	1
10	1	0	1	0	0	0	0
11	2	0	0	4	0	1	0
12	1	0	1	4	1	1	0
13	2	1	1	3	0	1	1
14	0	2	0	3	1	1	1
15	2	2	0	1	0	1	1
16	0	2	1	2	0	0	0
17	0	0	0	0	0	1	1
18	1	2	1	0	0	0	0
19	1	1	0	3	0	0	0
20	2	2	1	4	0	0	1
21	1	2	1	0	0	0	1
22	1	0	0	1	1	1	1

In [19]:
```
#fit training data attributes and labels.
clf=clf.fit(data1[['Occupied', 'Price', 'Music', 'Location', 'VIP', 'Favorite Beer']],data1[
'Enjoy'])
```

In [20]:
```
#here 'Yes' is mapped to 1 and 'No' is mapped to 0. so the tree predicted 'Yes' as Output
clf.predict(testing)
```

Out[20]:
```
array([1])
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [25]:
```python
from sklearn.tree import _tree

def dtfrom_func(clf, cols):
    dt = clf.tree_
    col=[]
    for f in dt.feature:
        col.append(cols[f])

    def rec(attr, d):
        indent = "   " * d
        if dt.feature[attr] != _tree.TREE_UNDEFINED:
            name = col[attr]
            threshold = dt.threshold[attr]
            print ("if",name," less than or equal to ",threshold," then")
            rec(dt.children_left[attr], d+1)
            print ("else")
            rec(dt.children_right[attr], d+1)
        else:
            print ("Leaf node: ",(dt.value[attr]))

    rec(0, 1)
```

In [27]:
```python
#Leaf nodes being number of 'Yes' and 'No' instances.
dtfrom_func(clf,data1.columns)
```

```
if Favorite Beer  less than or equal to  0.5  then
if Occupied  less than or equal to  1.5  then
if Location  less than or equal to  0.5  then
if Music  less than or equal to  0.5  then
Leaf node:  [[0. 1.]]
else
if Price  less than or equal to  1.0  then
Leaf node:  [[1. 0.]]
else
Leaf node:  [[1. 1.]]
else
Leaf node:  [[4. 0.]]
else
if Location  less than or equal to  2.5  then
if Location  less than or equal to  1.5  then
Leaf node:  [[0. 1.]]
else
Leaf node:  [[1. 0.]]
else
Leaf node:  [[0. 2.]]
else
if Location  less than or equal to  3.5  then
Leaf node:  [[0. 8.]]
else
Leaf node:  [[2. 0.]]
```

In [ ]:

In [ ]: