I. Data structure

To lessen the computing burden, I transfer the original night-out dataset from .txt file to .csv file. This .csv file can be stored into a 22 times 7 data frame in program. The seven columns are six attributes of data entities: Occupied, Price, Music, Location, VIP, Favorite Beer; as well as the class each entity belongs to —'Yes' for enjoy, 'No' for not so.

Additionally, I store all six attributes into a single variable—fs_of_enjoy—to run the program more readily; and the original class labels: "Yes" and "No" are also respectively changed to "1" and "0" for avoiding confusion with attribute values "Yes" and "No" in "VIP" and "Favorite Beer".

Thus, the data to run the decision tree is as follows:

In [1]:
```
import pandas as pd
import numpy as np
import math
```

In [2]:
```
#Read the csv.file, storing it to varaible "data_enjoy":

data_enjoy = pd.read_csv("C:/Users/User/Desktop/USC/Machine Learning/dt_data.csv")

#Replace the 'Enjoy' labels:"1" for "yes" and "0" for "no":

(data_enjoy['Enjoy']).replace(['Yes', 'No'], [1, 0], inplace=True)

#Save all attributes to variable fs_of_enjoy to run program with ease:

fs_of_enjoy = ['Occupied', 'Price', 'Music', 'Location', 'VIP', 'Favorite Beer']
```

	Occupied	Price	Music	Location	VIP	Favorite Beer	Enjoy
0	High	Expensive	Loud	Talpiot	No	No	0
1	High	Expensive	Loud	City-Center	Yes	No	1
2	Moderate	Normal	Quiet	City-Center	No	Yes	1
3	Moderate	Expensive	Quiet	German-Colony	No	No	0
4	Moderate	Expensive	Quiet	German-Colony	Yes	Yes	1
5	Moderate	Normal	Quiet	Ein-Karem	No	No	1
6	Low	Normal	Quiet	Ein-Karem	No	No	0
7	Moderate	Cheap	Loud	Mahane-Yehuda	No	No	1
8	High	Expensive	Loud	City-Center	Yes	Yes	1
9	Low	Cheap	Quiet	City-Center	No	No	0
10	Moderate	Cheap	Loud	Talpiot	No	Yes	0
11	Low	Cheap	Quiet	Talpiot	Yes	Yes	0
12	Moderate	Expensive	Quiet	Mahane-Yehuda	No	Yes	1
13	High	Normal	Loud	Mahane-Yehuda	Yes	Yes	1
14	Moderate	Normal	Loud	Ein-Karem	No	Yes	1
15	High	Normal	Quiet	German-Colony	No	No	0
16	High	Cheap	Loud	City-Center	No	Yes	1
17	Low	Normal	Quiet	City-Center	No	No	0
18	Low	Expensive	Loud	Mahane-Yehuda	No	No	0
19	Moderate	Normal	Quiet	Talpiot	No	No	1
20	Low	Normal	Quiet	City-Center	No	No	1
21	Low	Cheap	Loud	Ein-Karem	Yes	Yes	1

II. The Program:

1. First, I define the function of entropy:

   Inputs: ***data*** = data to calculate the entropy

   label=the class to categorize to, e.g. “Enjoy or not”

   Output: entropy value

```
In [3]: #First, define the entropy function:

        def get_entropy(data, label):
            label_value, label_count = np.unique(data[label], return_counts=True)

            entropy_value = np.sum(np.fromiter(((-label_count[i]/np.sum(label_count))*
                                                 np.log2(label_count[i]/np.sum(label_count))
                                          for i in range(len(label_value))),float))
            return (entropy_value)

        #Test the above fucntion with calculating total entropy of night-outs data
        get_entropy(data_enjoy,'Enjoy')

Out[3]: 0.9760206482366149
```

2. After this, define the function “info_gain” to get the information gain by subtracting the weighted entropy values of attributes from the whole dataset entropy:

   Inputs: ***data*** = data to calculate the entropy, get the information gain

   attribute = an attribute to calculating entropy

   label = the class the data entities to categorize to, e.g. “Enjoy or not” in our case

   Output: the information gain

```
In [24]: #Then,define the function of calculating information gain

         def info_gain(data, attribute, label):
             data_entropy = get_entropy(data, label)

             feat_value, feat_number = np.unique(data[attribute], return_counts=True)

         #feat_value = the feature values a certain feature (attribute) has,
         #feat_number = the numbers of every feature values

             weighted_feat_entropy = np.sum([(feat_number[i]/np.sum(feat_number))*
                                     get_entropy((data_enjoy.where(data_enjoy[attribute] == feat_value[i]).dropr
                                     for i in range(len(feat_value))])

             information_gain = data_entropy - weighted_feat_entropy

             return(information_gain)
```

3. The third function —“best_attri”—is to find the index in attribute list of attribute that returns the largest information gain:

   Inputs: ***data*** = data to calculate the entropy, get the information gain

   attributes = the attributes (features) of dataset

   label = the class the data entities to categorize to

   Outout: the index number of attribute that returns the largest information gain

```
In [26]: #Define the function to find the attribute that gives us the largest information gain

         def best_attri(data,attributes,label):

             for i in range(len(attributes)):
                 gain = info_gain(data,(attributes[i]),label)

                 best_attri_index = np.argmax(gain)

             return (best_attri_index)
```

4. Next, combining the above functions, we can construct the decision tree-building function—"create_tree":

   Inputs: ***data*** = data to calculate the entropy, get the information gain, and construct the decision tree

   attributes = the attributes (features) of dataset

   label = the class the data entities to categorize to

   root_node = the node the leaf nodes grow from, default it as None

   Output: decision tree

In "create_tree" function, I first set the conditions that will halt the tree constructing; that are, when all entities belong to the same class or all attributes have been used out.

```
In [20]: #Constructing the decision tree with above functions

def create_tree (data,attributes,label,root_node = None):

#If all entities belongs to the same label class, returning that label class:

    if len(np.unique(data[label])) <= 1:
        return np.unique(data[label])[0]

#If no attributes remained,returning root node:

    elif len(attributes) == 0:
        return root_node
```

If the data does not meet the above two conditions, keep building the tree:

```
#Elsewhile, keep constructing the decision tree:

    else:

#Update the root node value:
        root_node = np.unique(data[label])[np.argmax(np.unique(data[label],return_counts=True)[1])]

    best_attribute_number = best_split_feat(data,attributes, label)
#best_attribute_number: the index in attribute list of the attribute having the largest info gain

    best_attribute_name = attributes[best_attribute_number]
#best_attribute_name: that attribue's name


#Delete the used attributes from the attribute list:

    attributes = [k for k in attributes if k!= best_attribute_name]

#Define the decision tree's foundation:

    Decision_tree = {best_attribute_name:{}}
```

The tree building process is to calculate the best information gain, find the attribute of returning best information gain, reconstruct the dataset, then calculate the best information gain repeatedly.

```
#Start to build the tree:

    for i in np.unique(data[best_attribute_name]):

        i = i

#Reconstruct the data everytime we find a best-information gain attribute:
        new_data_df = data_enjoy.where((data[best_attribute_name])==i).dropna()

        Decision_tree[best_attribute_name][i] = create_tree(new_data_df,attributes,label,root_node)

    return Decision_tree
```

5. After the decision tree is completed, create a classifier to categorize the new data entities:

Inputs: ***my_tree*** = decision tree (the output of function "create_tree")

attrs_to_categorize = the attributes (features) of dataset

test_data = the attribute values of testing data

Output: the class the testing data entities belong to

```
In [22]: #Build the classifier

         def classifier(my_tree, attrs_to_categorize, test_data):

         #Because the data type of decision tree is dictionary,we use index to filter through the decision tree:
             Dict = my_tree[list(my_tree.keys())[0]]
             Index = attrs_to_categorize.index(list(my_tree.keys())[0])
             class_Label = '0'

             for key in Dict.keys():
                 if test_data[Index] == key:
                     if type(Dict[key]).__name__ == 'dict':
                         class_Label = classifier(Dict[key],attrs_to_categorize, test_data)
                     else:
                         class_Label = Dict[key]
             return class_Label
```

II. The Implement Result:

1. Run the create_tree function with the night-out dataset, the decision tree is as follows:

Input: ***Data*** = data_enjoy

Attributes = fs_of_enjoy

Label = 'Enjoy'

```
In [21]: create_tree(data_enjoy,fs_of_enjoy,'Enjoy',root_node = None)

Out[21]: {'Occupied': {'High': {'Price': {'Cheap': 1.0,
             'Expensive': {'Music': {'Loud': {'Location': {'City-Center': 1.0,
                 'Talpiot': 0.0}}}},
             'Normal': {'Music': {'Loud': 1.0, 'Quiet': 0.0}}}},
           'Low': {'Price': {'Cheap': {'Music': {'Loud': 1.0, 'Quiet': 0.0}},
             'Expensive': 0.0,
             'Normal': {'Music': {'Quiet': {'Location': {'City-Center': {'VIP': {'No': {'Favorite Beer': {'N
         o': 0.0}}}},
                 'Ein-Karem': 0.0}}}}}},
           'Moderate': {'Price': {'Cheap': {'Music': {'Loud': {'Location': {'Mahane-Yehuda': 1.0,
                 'Talpiot': 0.0}}}},
             'Expensive': {'Music': {'Quiet': {'Location': {'German-Colony': {'VIP': {'No': 0.0,
                   'Yes': 1.0}},
                 'Mahane-Yehuda': 1.0}}}},
             'Normal': 1.0}}}}
```

2. Classify the testing data:

Occupied = Moderate

Price= Cheap

Music = Loud

Location = City-Center

VIP = No

Favorite Beer = No

```
In [23]: tree_enjoy = create_tree(data_enjoy,fs_of_enjoy,'Enjoy',root_node = None)

         #Entering a case for prediciton:

         classifier(tree_enjoy,['Occupied', 'Price', 'Music', 'Location', 'VIP', 'Favorite Beer'],
                   ['Moderate','Cheap','Loud','City-Center','No','No'])

Out[23]: '0'
```

The classification result is ‘0’, which means we would not enjoy a good night in Jerusalem new near eve.

III. The Challenges and Code-level Optimization

The challenges I face include tracking data structure and how to design a recursive process in tree construction. Among these two, the solution of the latter one is the most critical code optimization in my program.

First of all, the data structure is a challenge because, during the programming process, I might at times lose the track of data shape; what results is me being mistaken about variables or functions to use. For example, in the first edition of my programming, I forget the latest data shape, then sliced out the entire column under a certain attribute after calculating information gains.

The second challenge is how to arrange a proper programming process. In the beginning, one great hindrance to me was how to recursive through the dataset every time I get an attribute of best info gain. To deal with this, my original programming concept was to write two functions: root node functions and leaf functions, then input the information gains to them to construct the decision roots and their leaves manually:

```
In [ ]: class node():
            def __init__(self,i):
                self.i = i
                self.sub = {}

            def __str__(self):
                return "Node number of column=%d" % self.i

        class leaf():
            def __init__(self,label):
                self.label = label

            def __str__(self):
                return "Leaf:label=%d" % self.label
```

Unfortunately, this process was a time-sucker, not to mention the errors it might bring every time I input the new info gains.

Accordingly, to program with more efficiency, I start to optimize the program that will self-recursive through the dataset. This approach must meet two demands: the ability to slice the dataset and that to calculate the new info gains with the after-slice dataset. Luckily, I finally get the solutions from functions "data. where" and "data. unique". The "data. unique" function can locate the feature values under the attributes having the best info gain, then assist data slicing. The "data. where" otherwise plays a big role in returning the class labels for the after-slice dataset, which makes it easier to calculate the new dataset's entropy values:

```
#Start to build the tree:

    for i in np.unique(data[best_attribute_name]):

        i = i

#Reconstruct the data everytime we find a best-information gain attribute:
        new_data_df = data_enjoy.where((data[best_attribute_name])==i).dropna()

        Decision_tree[best_attribute_name][i] = create_tree(new_data_df,attributes,label,root_node)

    return Decision_tree
```

Thus, my code-level optimizations are primarily to update an auto recursive approach. The others are the aforementioned ones: assigning variable "fs_of_enjoy" as well as changing the representations of the class label to "1" and "0".