# DSCI 552: Machine Learning for Data Science

## Programming Assignment 1: Decision Trees

**Team members: Li An, Shengnan Ke**

**Due Date: 02/05/2022**

## Contribution:

We both reviewed the lecture that discussed the idea of decision trees. Next, we did research on how to construct a decision tree from scratch. However, we found that most of the examples online were using the library functions. Therefore, we found pseudocode that describes the process of constructing a decision tree. We followed the pseudocode, listed out functions and then divided the parts that worked on different functions individually.
**Li An: Part 1 (Choose the best feature to split, Split Data, Build the Decision tree), Part 2**
**Shengnan Ke: Part 1 (Data importation,Calculate the entropy, Majority Rule), Part 3**
After putting everything together, we start to debug and try to optimize the performance. Next, we start to take in the data for prediction. After we successfully print out the decision tree and work out the prediction. We start to edit the report and add helpful comments on the code.

## Part 1: Implementation: Building Decision Tree

**Load data:**
We made the following changes to the dt_data.txt file to make the data importation easier :

1. We removed the count number at the beginning of each row. “01,02…”
2. We removed the semicolon at the end of each row. “;”
3. We removed the feature list from the top line. “(Occupied, Price, …, Enjoy)”

We used the ***pandas*** package in python to help us load the dataset. After making the changes mentioned above, we can use the ***pandas.read_csv*** function to load the data file into the pandas dataframe, and add column names to the dataframe.

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

**Calculate the entropy [`calEntropy(dataset)`]:**
The first function is to calculate the entropy based on the following formula, where ***Pi*** means label ***i***'s frequency in the dataset ***S***.

$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$

Therefore in this function, firstly, we used a loop to calculate each kind of label's frequency, then made the summation in the second loop to get the entropy.

**Choose the best feature to split [`BestFeatureToSplit(dataset:pd.DataFrame)`]:**
This is the key function to implement the decision tree. Based on the idea of information gain, we know that the greater the information gain the less the impurity.
In this function, we went through each of the remaining features. For each feature, we call the ***calEntropy*** function to calculate the entropy of every possible value of the feature and then find the information gain based on the weighted average of entropy. After this step, we can find out the best feature with the maximum information gain.

**Split Data [`splitData(dataset, feat, value)`]:**
This function is helping us to split the dataset into pieces based on the best feature we got in the ***BestFeatureToSplit*** function. We not only filtered rows by possible values of the feature we chose to split on, but also removed the feature we have already used, making sure we are not looking back at the features used to split on.
For example, based on a dataset, we found that the "root node" is "Occupied" which means on the next step we will check if it is "High", "Moderate", or "Low". If in this instance, it is "High", next, we should go through all the other features "Price, Music, Location, VIP, Favorite Beer" and see which feature will be the next "node" to split on, except "Occupied".

**Majority Rule [`countMajority(labelList)`]:**
This function is for the situation after all the features are traversed, the label with the highest number of samples will be selected which is following the **Majority rule.** When we have more than one maximum (when there is no such thing as descending order), we randomly select one label from the equally large number as the final selection. This is one of the optimizations we have performed in our code.
Our logic here is:

1. First, the compValue stands for compare value. ***data.value_counts()*** rank the label list in a descending order. So the first element in the result list tells us the label with the most samples. For general cases, this label will be selected.
2. **Optimizations**: we realize that it is possible for more than one label to have the same amount of samples. At that time, we couldn't just pick the first label by default. Therefore, we decided to create a new list called newDataList to save labels with the same amount of labels. Then randomly selected one from the list. To make our prediction more accurate with this extra randomness.

**Build the Decision tree [`buildDecisionTree(ori_data, dataset)`]:**

First, there are four situations where we can reach the "leaf node" of the decision tree.

1. If the samples in the remaining dataset are all from the same label. We return it as the final result.
2. If all the features have been traversed. There are no more features for us to choose from. At this time we call the function ***countMajority*****,** use this to find out the final label.
3. There are some contradictory sample cases in the training data where all the features are the same (for instance, two samples, they both expensive, low occupied, good beer, loud music, located at City-Center, but one has labeled "enjoyed" another one has labeled "not enjoyed"). In this case, we also make this situation obey the majority rule.
4. As we kept splitting the dataset, sub-datasets would become smaller and smaller, even becoming an empty dataset at some nodes. For this situation, we took the majority label of the dataset before that split node as the result.

Following is the pseudo-code for this function:

```
def buildDecisionTree (dataset):
    if all samples in the dataset have the same label:
        return the label as the result
    if there are no more features can be used to split the dataset:
        return the majority label of the dataset
    find the best feature to split on based on information gain criteria
    if cannot find a feature to improve the entropy:
        return the majority label of the dataset
    else:
        add a new node for the best feature
        for each possible value of the best feature:
            split the dataset based on the value
            if the sub-dataset is empty:
                return the majority label of the dataset
            else:
                buildDecisionTree (sub-dataset)
```

**Challenges:**

1. As it requires us to use each feature as the split node only one time, we need to keep track of the remaining features for each node. We came to the idea of achieving this goal in the data splitting step.
2. We once had an error in the program when getting the possible values of the best feature. After debugging, we found that it's because the program cannot find the "best feature", so it returned nothing. Then we printed out the sub-dataset at where it had the error, finding that it was due to the contradictory samples in the training data.

**Print the decision tree and make the prediction:**

Following is the decision tree that we learned from the training dataset. We use indentations to indicate levels of the tree, and each line refers to the decision rule of one node.

Based on the decision tree we got, our prediction for the test data is: **“Yes, we will have a good night-out in Jerusalem for the coming New Year's Eve”**.

```
-Occupied: High
	-Location: Talpiot
		-result: No
	-Location: City-Center
		-result: Yes
	-Location: German-Colony
		-result: No
	-Location: Ein-Karem
		-result: Yes
	-Location: Mahane-Yehuda
		-result: Yes
-Occupied: Moderate
	-Location: Talpiot
		-Price: Expensive
			-result: No
		-Price: Normal
			-result: Yes
		-Price: Cheap
			-result: No
	-Location: City-Center
		-result: Yes
	-Location: German-Colony
		-VIP: No
			-result: No
		-VIP: Yes
			-result: Yes
	-Location: Ein-Karem
		-result: Yes
	-Location: Mahane-Yehuda
		-result: Yes
-Occupied: Low
	-Location: Talpiot
		-result: No
	-Location: City-Center
		-Price: Expensive
			-result: No
		-Price: Normal
```

```
                -result: No
        -Price: Cheap
                -result: No
-Location: German-Colony
        -result: No
-Location: Ein-Karem
        -Price: Expensive
                -result: Yes
        -Price: Normal
                -result: No
        -Price: Cheap
                -result: Yes
-Location: Mahane-Yehuda
        -result: No
```

## Part 2: Software Familiarization

One of the most commonly used libraries for machine learning is ***sklearn***, and the ***DecisionTreeClassifier*** offered in sklearn is a class capable of performing multi-class classification on a dataset.
To use it, we can simply create an instance of ***DecisionTreeClassifier***, and then call the fit function to fit the model. This classifier takes two parameters: an array X, sparse or dense, of shape `(n_samples, n_features)` holding the training samples, and an array Y of integer values, shape `(n_samples,)`, holding the class labels for the training samples. After being fitted, we can then call the predict function to predict the class of samples.
As for the print of the tree, sklearn offers several ways of printing trees, either by the ***plot_tree*** function, or by the ***export_graphviz*** exporter, or by the function ***export_text***.

Comparing the ***DecisionTreeClassifier*** classifier against our own implementation, we found several improvements we can do to our code:

1. Take other types of features into account. Our implementation can only deal with categorical features, but doesn't support numerical features. Even if ***DecisionTreeClassifier*** doesn't support categorical features directly, users can convert categorical features into numerical ones that the classifier supports, like one-hot encoding or dummy variables.
2. Make a more elegant print of trees, and provide more print options for users.
3. Add pruning steps. ***DecisionTreeClassifier*** provides options to add restrictions to the decision tree in the parameters, such as the maximum depth of the tree, the minimum number of samples required to split an internal node, and the minimum number of samples required to be at a leaf node.

4. ***DecisionTreeClassifier*** constructs binary trees using the feature and threshold that yield the largest information gain at each node, while our code constructs multiway trees, which is more computationally costly.

## Part 3: Applications

Decision trees are used in a wide range of applications in many aspects of life, in engineering, civil planning, law, finance, medical-related problems, and many other areas. Also, the decision tree can always “collaborate” with other techniques to build greater learning algorithms. For example(I learned it from an article), the researchers combined decision trees and artificial neural networks (ANN) on a difficult pharmaceutical data mining (KDD) drug discovery application. They were specifically predicting inhibition of a P450 enzyme which I believe similar combinations could also test and discover other types of medical products. Another example is ADL recognition in a smart home system. More specifically, it’s like a user’s activity recognition systems, based on an activity frame and a predictor based on the decision tree. In this case, this robot is made for elderly care, by recognizing elder people’s daily activities to predict their general next action. Once some unusual actions are detected, this device will send warnings to elder people's emergency contacts to check on their safety conditions.

## Code:

```
# DSCI 552: Machine Learning for Data Science
# Programming Assignment 1: Decision Trees
# Team members: Li An, Shengnan Ke

import math
import pandas as pd
import numpy as np
import random

def calEntropy(dataset):
   dataset = np.array(dataset)
   data_len = len(dataset)
   # LabelCount: number of samples for each label category
   labelCount = {}

   for row in dataset:
       label = row[-1]
       if label not in labelCount.keys():
           labelCount[label] = 0
       labelCount[label] += 1
```

```
    result = 0
    for key in labelCount.keys():
        prob = labelCount[key]/data_len # Frequency
        result -= prob*math.log2(prob)
    return result


# Choose the best feature to split based on the information gain criteria
def BestFeatureToSplit(dataset:pd.DataFrame):
    features = dataset.columns.values[0:-1]
    best_feat = ''
    max_Gain = 0
    for feat in features:
      # Entropy before split
      entropy = calEntropy(dataset)
      # Records the information gain of "feat"
      feat_values = dataset[feat].unique()
      # Initialize it with the max IG it can reach, and do subtraction from it in the
loop
      feat_Gain = entropy

      for val in feat_values:
        # data_v is the sub dataset when feat==val
        data_v = dataset[dataset[feat]==val]
        # ratio_v is the ratio of data_v to dataset
        ratio_v = (data_v.shape[0])/(dataset.shape[0])
        tmp_entropy = calEntropy(data_v)
        feat_Gain -= ratio_v*tmp_entropy

      # Print('gain: ',feat_Gain,'\n')
      if feat_Gain > max_Gain:
        max_Gain = feat_Gain
        best_feat = feat
    return max_Gain, best_feat


# Split the dataset based on the feature chosen in BestFeatureToSplit
def splitData(dataset, feat, value):
    feat_values = dataset.columns.values
    ret_feats = [f for f in feat_values if f != feat]
    # Remove the used features
    ret_dataset = dataset.loc[dataset[feat]==value, ret_feats]
```

```
    return ret_dataset


def countMajority(labelList):
    '''
    labelList: only the label column of the dataset
    '''
    data = pd.Series(labelList)
    compValue = data.value_counts()[0] # gives the label with maximun number of samples
    newDataList = [] # to store labels with same amount of samples
    newDataList.append(data.value_counts().index[0])
    for i in range(1,len(data.value_counts())):
      if compValue == data.value_counts()[i]:
        newDataList.append(data.value_counts().index[i])
    return random.choice(newDataList) # choose one label from the newDataList randomly


def buildDecisionTree(ori_data, dataset):#ori_data: the original dataset with
everything in it
    labelList = list(dataset['Enjoy'])

    # when samples in dataset all have the same label, stop recursion
    if labelList.count(labelList[0]) == len(labelList):
      return labelList[0]

    # all features are used up, no more features can be used to split on
    if len(dataset.columns.values) == 1:
      return countMajority(labelList)

    # find out the best feature to split on
    _, bestFeat = BestFeatureToSplit(dataset)

    # if there are contradictory samples (having same value on all features,
    # but have different labels), then follow the majority rule
    if bestFeat == '':
      return countMajority(labelList)

    # As we split the data, the sub dataset may not cover all the values of the
feature.
    # So here ori_data must be the original dataset which haven't been split
    feat_values = ori_data[bestFeat].unique()
```

```
    decisionTree = {bestFeat:{}}  # add a branch

    # split the dataset
    for val in feat_values:
      sub_dataset = splitData(dataset, bestFeat, val)
      if len(sub_dataset) == 0:
        decisionTree[bestFeat][val] = countMajority(labelList)
      else:
        decisionTree[bestFeat][val] = buildDecisionTree(ori_data, sub_dataset)

    return decisionTree

# Load data
df_enjoy = pd.read_csv('dt_data.txt', sep=", ", header = None)
df_enjoy.columns = ["Occupied", "Price", "Music", "Location", "VIP", "Favorite Beer",
"Enjoy"]


# Training
myTree = buildDecisionTree(df_enjoy, df_enjoy)
print(myTree)


# Prediction

# predict values:
test_data = {'Occupied': 'Moderate', 'Price': 'Cheap', 'Music': 'Loud', 'Location':
'City-Center', 'VIP': 'No', 'Favorite Beer': 'No'}

key = next(iter(myTree))
result = myTree
while isinstance(result, dict):
 val = test_data[key]
 result = result[key][val]
 key = next(iter(result))

print(result)
```