Chuxin Zhang, Wei Huang
DSCI 552
Professor Satish Kumar Thittamaranahalli
20 November, 2020

Decision Tree Report

We produced five functions to handle the Decision Tree problem. They are used to calculate the entropy, choose the best feature as root, divide the sub dataset, create the tree, visualize the decision tree as graph and predict test data. Our decision tree will show as a graph. This report will describe each function line by line and also include some problems we found in the process.

Contribution

- Programming:
  - Collaboration Part: We figure out the main program together including:
    - Calculating the Entropy
    - Select the best feature
    - Divide the sub dataset
    - Create the tree
  - Individual Part
    - Wei Huang: Visualization the decision tree
    - Chuxin Zhang: Prediction
- Report:
  - In the report, Chuxin Zhang writes the parts of:
    - Calculating the Entropy
    - Choose Best Feature
    - Dividing Sub dataset
    - Create Tree and print the tree
    - Prediction
  - In the report, Wei Huang writes the parts of:
    - Visualization of decision tree
    - Problems we meet

Calculating the Entropy

```
37 def calcEntropy(dataset):
38     numdata = len(dataset)
39     labelCounts = {}
40     for row in dataset:
41         currentLabel = row[-1]
42         if currentLabel not in labelCounts.keys():
43             labelCounts[currentLabel] = 0
44         labelCounts[currentLabel] += 1
45     entropy = 0
46     for i in labelCounts:
47         pro = labelCounts[i]/numdata
48         entropy += pro * log(1/pro,2)
49     return entropy
```

This function includes the process of calculating the entropy. We firstly count the amount of labels of this dataset. Then calculating entropy according to the probability of label counts divided by the data length. This function would return a float of entropy.

Choose Best Feature

```
63 def chooseBest(dataset):
64     numFeature = len(dataset[0])-1
65     baseEnt = calcEntropy(dataset)
66     bestGain = 0
67     bestFeature = -1
68     for i in range(numFeature):
69         featurelist = []
70         for val in dataset:
71             featurelist.append(val[i])
72         uniqueVals = set(featurelist)
73         newEnt = 0
74         for val in uniqueVals:
75             subData = subDataset(dataset,i,val)
76             prob = len(subData)/len(dataset)
77             newEnt += prob*calcEntropy(subData)
78         infoGain = baseEnt - newEnt
79         if (infoGain > bestGain):
80             bestGain = infoGain
81             bestFeature = i
82     return bestFeature
```

After calculating the entropy, we need a function to decide the best gain and choose the corresponding feature. We set zero for default value of best gain, and when there is a higher value of info gain appeared, the value of best gain will be replaced and the index for this attribute will also be stored. Finally, this function will return a float for the index of the feature which has the best information gain.

Dividing Sub dataset

```
51 def subDataset(dataset,index,value):
52     subData = []
53     for subrow in dataset:
54         if subrow[index] == value:
55             reduceSub = subrow[:index]
56             reduceSub.extend(subrow[index+1:])
57             subData.append(reduceSub)
58     return subData
```

This function is used to divide the dataset when we know the index of best feature and corresponding value. And since we don't need the index attribute we input, the function will generate a sub dataset without the best feature. This function would return a list of sub data.

Create Tree (Main Function)

```
84 def createTree(dataset,labels):
85     classlist = [example[-1] for example in dataset]
86     if classlist.count(classlist[0]) == len(classlist):
87         return classlist[0]
88     if len(dataset[0]) == 1:
89         return max(classlist,key=classlist.count)
90     bestFeat = chooseBest(dataset)
91     if bestFeat == -1:
92         return max(classlist,key=classlist.count)
93     bestFeatLabel = labels[bestFeat]
94 
95     myTree = {bestFeatLabel:{}}
96     featval = [i[bestFeat] for i in dataset]
97     uniqueVals = set(featval)
98 
99     for val in uniqueVals:
100        sublabel = labels[:bestFeat]
101        sublabel.extend(labels[bestFeat+1:])
102        myTree[bestFeatLabel][val] = createTree(subDataset(dataset,bestFeat,val),sublabel)
103    return myTree
```

So now we have already defined all the functions we need, we can start to program our main function to create a decision tree. Firstly we will handle some specific situations. If at this time the data we input has only value in the label, which means we can directly output the leaf node, it will return the label value. And if the data is split into only one column and still can't get a pure leaf node, we will simply decide the majority of labels.

After handling these two specific situations, we can run the bestFeat function to decide the index of this dataset's best attribute. At this time, we find a special case after getting the best feature. The best feature could be the default value (-1) we set in the program when the amount of True is exactly the same as the amount of False. We will discover this problem later in the later part. In brief, we will also choose either value as a valid leaf node.

Getting the index and name of the root node, we then store its values in a list. And for each value of the attribute, we've looped a regression to generate the whole tree.

Graph

Construct the annotation tree

```
7   def getLeafs(tree):
8       leaves = 0
9       firstStr = list(tree.keys())[0]
10      secondDict = tree[firstStr]
11      for key in secondDict.keys():
12          if type(secondDict[key]).__name__=='dict':
13              leaves += getLeafs(secondDict[key])
14          else:
15              leaves += 1
16      return leaves
```

First, we iterate over the tree. And we will determine whether it is a dictionary type. If yes, we will do the recursion to visit this dictionary. If not, we will return the leaf node. In this way, we calculate the number of leaf nodes in the decision tree through recursion, and return the number of nodes.

```
18  def getDepth(tree):
19      depth = 0
20      firstStr = list(tree.keys())[0]
21      secondDict = tree[firstStr]
22      for key in secondDict.keys():
23          if type(secondDict[key]).__name__=='dict':
24              currentDepth = 1 + getDepth(secondDict[key])
25          else:
26              currentDepth = 1
27          if currentDepth > depth:
28              depth = currentDepth
29      return depth
```

In the same recursion way, we can get the depth of the tree.

Draw the structure of the tree

```
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    createPlot.ax1.annotate(nodeTxt, xy=parentPt, xycoords='axes fraction',
                            xytext=centerPt, textcoords='axes fraction',
                            va="center", ha="center", bbox=nodeType, arrowprops=arrow)
```

Print the label texts in the node.

```
44  def plotMidText(cntrPt, parentPt, txtString):
45      xMid = (parentPt[0] - cntrPt[0]) / 2.0 + cntrPt[0]
46      yMid = (parentPt[1] - cntrPt[1]) / 2.0 + cntrPt[1]
47      createPlot.ax1.text(xMid, yMid, txtString, va="center", ha="center", rotation=30)
```

Print the value of the label in the middle of the arrow.

```python
def plotTree(tree, parentPt, nodeTxt):
    numLeafs = getLeafs(tree)
    depth = getDepth(tree)
    firstStr = list(tree.keys())[0]
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs)) / 2.0 / plotTree.totalW, plotTree.yOff)
    plotMidText(cntrPt, parentPt, nodeTxt)
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
    secondDict = tree[firstStr]
    plotTree.yOff = plotTree.yOff - 1.0 / plotTree.totalD
    for key in secondDict.keys():
        if type(secondDict[
                    key]).__name__ == 'dict':
            plotTree(secondDict[key], cntrPt, str(key))
        else:
            plotTree.xOff = plotTree.xOff + 1.0 / plotTree.totalW
            plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff), cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
    plotTree.yOff = plotTree.yOff + 1.0 / plotTree.totalD
```

First, we will get the number of leaf nodes and the depth of the tree, which will determine the length and width of the tree. And then we get the feature of the first node and its values in the current tree. Then draw them.

Next step, we iterate over its child nodes and make judgments. If the node is a dictionary then we do the recursion to visit the node. Otherwise, it is a leaf node, and we will draw the leaf.

After repeating these procedures until we've visited all the nodes, we will get the picture output of the tree.

Print the Tree

```python
dataset = [['High','Expensive','Loud','Talpiot','No','No','No'],
           ['High','Expensive','Loud','City-Center','Yes','No','Yes'],
           ['Moderate','Normal','Quiet','City-Center','No','Yes','Yes'],
           ['Moderate','Expensive','Quiet','German-Colony','No','No','No'],
           ['Moderate','Expensive','Quiet','German-Colony','Yes','Yes','Yes'],
           ['Moderate','Normal','Quiet','Ein-Karem','No','No','Yes'],
           ['Low','Normal','Quiet','Ein-Karem','No','No','No'],
           ['Moderate','Cheap','Loud','Mahane-Yehuda','No','No','Yes'],
           ['High','Expensive','Loud','City-Center','Yes','Yes','Yes'],
           ['Low','Cheap','Quiet','City-Center','No','No','No'],
           ['Moderate','Cheap','Loud','Talpiot','No','Yes','No'],
           ['Low','Cheap','Quiet','Talpiot','Yes','Yes','No'],
           ['Moderate','Expensive','Quiet','Mahane-Yehuda','No','Yes','Yes'],
           ['High','Normal','Loud','Mahane-Yehuda','Yes','Yes','Yes'],
           ['Moderate','Normal','Loud','Ein-Karem','No','Yes','Yes'],
           ['High','Normal','Quiet','German-Colony','No','No','No'],
           ['High','Cheap','Loud','City-Center','No','Yes','Yes'],
           ['Low','Normal','Quiet','City-Center','No','No','No'],
           ['Low','Expensive','Loud','Mahane-Yehuda','No','No','No'],
           ['Moderate','Normal','Quiet','Talpiot','No','No','Yes'],
           ['Low','Normal','Quiet','City-Center','No','No','Yes'],
           ['Low','Cheap','Loud','Ein-Karem','Yes','Yes','Yes']]

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

```python
myTree = createTree(dataset,labels)
print(json.dumps(myTree,indent=4, ensure_ascii=False))
createPlot(myTree)
```

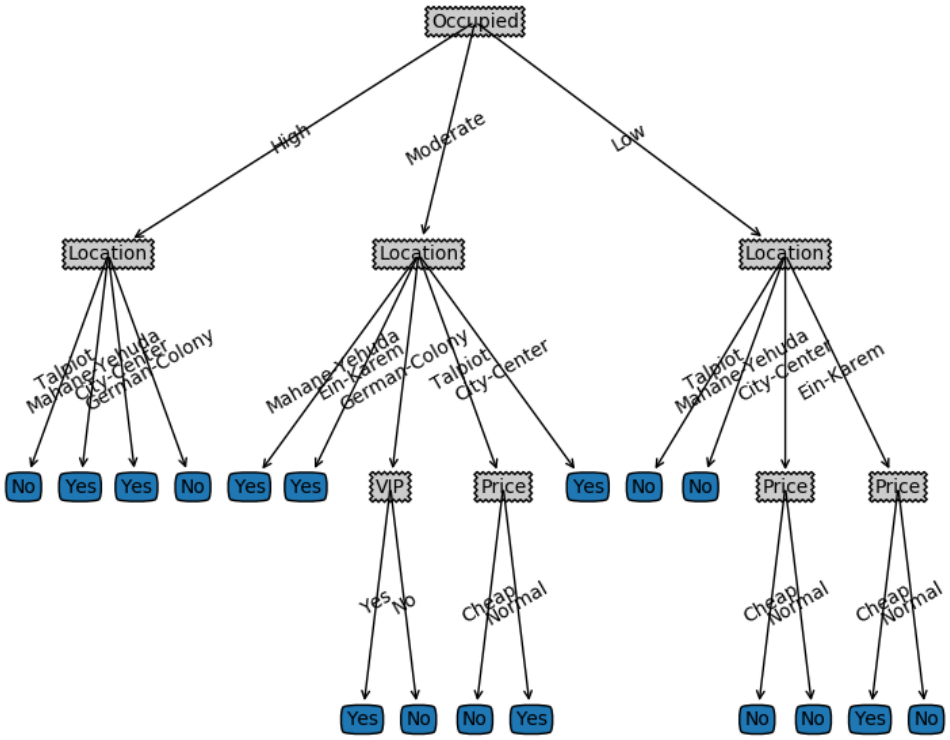
Right now, we can finally generate our tree. We could simply run the createTree function and the tree will be generated as a dictionary type. At this time we can already print the tree as dict like below. Since this print way is not quite clear, we then use the createPlot function to visualize the tree.

Our final tree is printed as:

1) Dictionary type:


```
{
    "Occupied": {
        "High": {
            "Location": {
                "City-Center": "Yes",
                "Talpiot": "No",
                "German-Colony": "No",
                "Mahane-Yehuda": "Yes"
            }
        },
        "Moderate": {
            "Location": {
                "German-Colony": {
                    "VIP": {
                        "No": "No",
                        "Yes": "Yes"
                    }
                },
                "Talpiot": {
                    "Price": {
                        "Cheap": "No",
                        "Normal": "Yes"
                    }
                },
                "Mahane-Yehuda": "Yes",
                "City-Center": "Yes",
                "Ein-Karem": "Yes"
            }
        },
        "Low": {
            "Location": {
                "City-Center": {
                    "Price": {
                        "Cheap": "No",
                        "Normal": "No"
                    }
                },
                "Talpiot": "No",
                "Ein-Karem": {
                    "Price": {
                        "Normal": "No",
                        "Cheap": "Yes"
                    }
                },
                "Mahane-Yehuda": "No"
            }
        }
    }
}
```

2) Graph



Prediction

```
In [209]: def predict(myTree,labels,testdata):
     ...:     root = list(myTree.keys())[0]
     ...:     subtree = myTree[root]
     ...:     featindex = labels.index(root)
     ...:     for key in subtree.keys():
     ...:         if testdata[featindex] == key:
     ...:             if type(subtree[key]).__name__ == 'dict':
     ...:                 outcome = predict(subtree[key],labels,testdata)
     ...:             else:
     ...:                 outcome = subtree[key]
     ...:     return outcome
     ...:
     ...:
     ...: testdata = ['Moderate','Cheap','Loud','City-Center','No','No']
     ...:
     ...: print(predict(myTree,labels,testdata))
Yes
```

In the prediction function, we also use regression to handle our request. We will firstly get the root node and subtree under the root. If the type of subtree is still the dictionary but not the leaf node, we will then re-run the predict function to do the regression algorithm until outputting the final result.

In the screenshot, we could see our predicted result for test data is **"Yes"**.

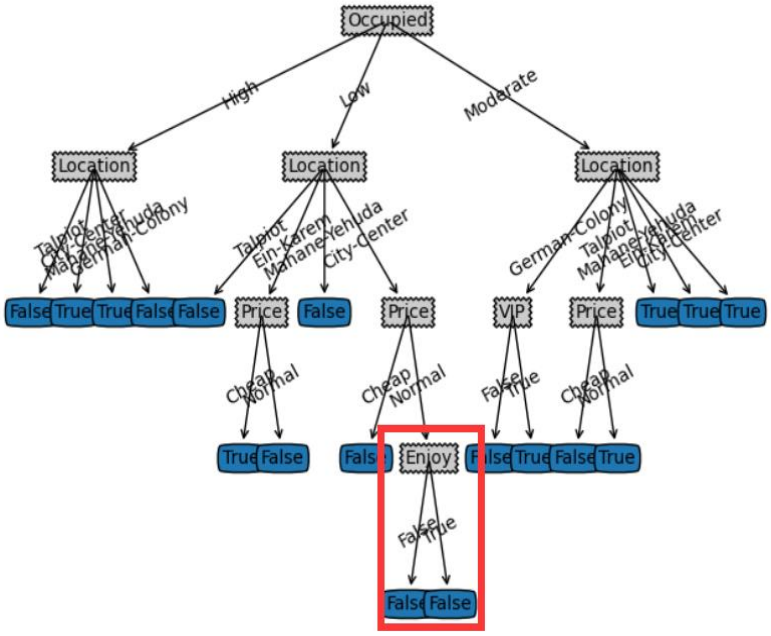
Problems

```
84  def bestDvd(dataSet):
85      sum = len(dataSet[0]) - 1
86      # print(sum)
87      baseEntropy = calcEntropy(dataSet)
88      bestGain = 0.0
89      bestFeature = 0
```

At first, in the function of choosing the best feature, we initialized the tag used to record the best feature as 0. And we use our simple test data set to run the program. We get a wrong output. After debugging, we found the initial value of this tag cannot be 0, because labels[0] represents the first label in the tree.

```
89      bestFeature = -1
```

Then we change the initial value to -1. Then we ran our test dataset and got the right output. However, when we switch to the given dataset, we get the result below:



In the red frame marked, we found our result label appeared as a judge label, which should not happen. After checking the difference between two datasets, we found these two data.

```
['Low', 'Normal', 'Quiet', 'City-Center', False, False, False],
['Low', 'Normal', 'Quiet', 'City-Center', False, False, True],
```

These data have totally the same features, but different results. Which means when we split the dataset to the end, the final dataset is still not a pure data. And in this condition, the program will proceed to the next split. But there is no judge label anymore, so the program will wrongly choose the result label to do the split and get the error output.

After tracking the data, we found that in our choosing feature function, if there is no feature to choose, it will not change the feature which means the function will return the initial value -1.

```
99          if(Gain > bestGain):
100             bestGain = Gain
101             bestFeature = i
102     return bestFeature
```

However, the initial value will never appear in a normal procedure. So when the function returns -1, we can judge that the dataset has been splitted to the end and it is still impure. Then there comes the next question: How should we deal with the impure data? We tried this solution:

```
118     BestFeature = bestDvd(dataSet)
119     list = [example[-1] for example in dataSet]
120     if BestFeature == -1:
121         return max(list,key=list.count)
```

If the returned feature is -1, we will return the most frequent result, which means our result will obey the majority.

If the frequency of every result is equal, this function will return the first result. In this condition, we believe that the dataset is insufficient and need supplement.