INF 552 Programming Assignment 1: Decision Trees

4472-6221-33 Yo Shuan Liu

I. Introduction

I wrote this homework in Python. I currently don't have any group partner, so I am writing this homework on my own this time, but I am open to pair up with any classmate from different major.

II. Implementation

Data Structure

There are two important components in my decision tree, one is decision node, another is leaf node.

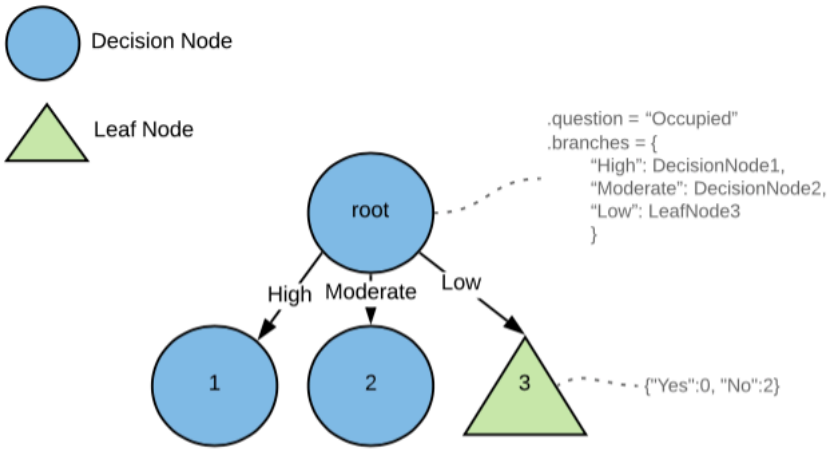
A decision node asks a question, it holds:

(1) a reference to the question, which is the column number of attribute,

(2) and decision nodes or leaf nodes after each possible answer of this question, which is stored as a dictionary with keys equal to the answer and values as the next layer of decision node or leaf node. The number of key-value pairs in this dictionary depends on how many unique value this attribute has in this data subset.

A leaf node classifies data. It holds a dictionary with keys "Yes" and "No", which is the label of our dataset, and the corresponding values are the number of times "Yes" and "No" appears in the rows from the training data that reach this leaf.

I draw the diagram below for better illustrating the components and data structure of my decision tree, however, this is not the final output I obtain.



There is an important procedure in my decision tree learning algorithm, splitting the dataset. To be able to build decision node mentioned above, I have to indicate how this subset is derived from previous data subset. That is, make a dictionary to store these data subsets, with the unique value of the attribute as key and the data subsets as value. For example, the training data is split after we learned which question to ask at root decision node, and the split data subsets are stored as follow:

{"High": [[data subsets with "Occupied" == "High"]], "Moderate": [[data subsets with "Occupied" == "Moderate"]], "Low": [[data subsets with "Occupied" == "Low"]]}

By taking advantage of storing data subsets in this way, I can build branches in decision node instance (branches in root decision node) after finding out the most critical question for that data subset (question to ask in decision node 1).

Code Level Optimization

1. Before:
   At first, I let the algorithm go through each attribute every time to find out the most valuable question. It turns out that after one question is asked, then it won't be asked again later because splitting data by used attribute along this path won't decrease the entropy.
   After:
   Therefore, I created a set which contains unused attribute along the path, I called it **remainingAttrSet**, and make it as one of the input of my recursive function when building the tree.
2. Before:
   Initially, I took advantage of data frame structure of "pandas", but I later found out that I cannot retrieve a column in pandas' data frame using the column index (I think it is because panda's data frame uses a hash map concept). In order to follow pandas' data structure, I had to store column names not only in my decision node instance but also the remainingAttrSet.
   After:
   Finally, I decided to get rid of pandas and store my data in a 2x2 matrix form, so I could indicate an attribute by its column index, and column names are used only when I print out my tree, which is more effective than storing/retrieving the column name all the time.
3. I also organized my code in an object-oriented manner, which contains three classes, DecisionNode, LeafNode, and DecisionTree. All functions related to building the decision tree is in DecisionTree class.

Challenge I Face

How to store multiple branches is a challenge at first, because unlike binary search tree which has at most two child nodes, the number of child node of my decision tree depends on the number of unique value of an attribute. And how to store children for each decision node by not indicating a fix number of children?
Finally, I solved this obstacle by storing branches as a python dictionary mentioned in the data structure part above. Therefore, the test data would know which is the next decision node they have to follow after answering one question, and python dictionary also provides a flexibility on the number of key-value pairs which allows me to store whatever amount of child node the decision node have.

Tree Printing Format

I print my decision tree using a recursive function with for loop. For loop allows me to loop through every child node and print every node recursively until a leaf node is reached.

1. “Q: …?” indicates the question asked by a decision node.
2. And all possible answer for that question forms branches.
3. “Predict {‘No’: ..., ‘Yes’: ...}” indicates a leaf node is reached.
4. I also use “_ _ _ _ _ _ _” to indicate the subtree rooted at the node has been fully grown.

Please find my decision tree as follow:

```
Q: Occupied?
 |-- High
 |   Q: Location?
 |    |-- Talpiot
 |    |    Predict {'No': 1, 'Yes': 0}
 |    |-- City-Center
 |    |    Predict {'No': 0, 'Yes': 3}
 |    |-- Mahane-Yehuda
 |    |    Predict {'No': 0, 'Yes': 1}
 |    |-- German-Colony
 |    |    Predict {'No': 1, 'Yes': 0}
 |    |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 |
 |-- Moderate
 |   Q: Location?
 |    |-- City-Center
 |    |    Predict {'No': 0, 'Yes': 1}
 |    |-- German-Colony
 |    |    Q: VIP?
 |    |     |-- No
 |    |     |    Predict {'No': 1, 'Yes': 0}
 |    |     |-- Yes
 |    |     |    Predict {'No': 0, 'Yes': 1}
 |    |     |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 |    |
 |    |-- Ein-Karem
 |    |    Predict {'No': 0, 'Yes': 2}
 |    |-- Mahane-Yehuda
 |    |    Predict {'No': 0, 'Yes': 2}
 |    |-- Talpiot
 |    |    Q: Price?
 |    |     |-- Cheap
 |    |     |    Predict {'No': 1, 'Yes': 0}
 |    |     |-- Normal
 |    |     |    Predict {'No': 0, 'Yes': 1}
 |    |     |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 |    |
 |    |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 |
 |-- Low
 |   Q: Location?
 |    |-- Ein-Karem
 |    |    Q: Price?
 |    |     |-- Normal
 |    |     |    Predict {'No': 1, 'Yes': 0}
 |    |     |-- Cheap
 |    |     |    Predict {'No': 0, 'Yes': 1}
 |    |     |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 |    |
 |    |-- City-Center
 |    |    Q: Price?
 |    |     |-- Cheap
 |    |     |    Predict {'No': 1, 'Yes': 0}
 |    |     |-- Normal
 |    |     |    Predict {'No': 1, 'Yes': 1}
 |    |     |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 |    |
 |    |-- Talpiot
 |    |    Predict {'No': 1, 'Yes': 0}
 |    |-- Mahane-Yehuda
 |    |    Predict {'No': 1, 'Yes': 0}
 |    |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 |
 |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Prediction

My prediction for (occupied = Moderate; price = Cheap; music = Loud; location = City-Center; VIP = No; favorite beer = No) is **YES**.

III. Software Familiarization

Python Library

sklearn.tree.DecisionTreeClassifier

Python's scikit-learn library allows us to conduct decision tree algorithm. For decision tree classifier in scikit-learn, we can choose whether to use information gain or Gini impurity to measure the split quality. We can also indicate the maximum depth of the tree as well as the minimum number of samples in each leaf node, and many other parameters for preventing the decision tree from overfitting.

To be Improved: overfitting the training dataset

I realize that I didn't include any features to prevent overfitting in my algorithm. Since the dataset provided in this homework is rather small, my decision tree has maximum depth = 3, and my smallest leaf has only one sample in it, which might result in overfitting the training dataset and a high out-of-sample error.

IV. Application

One interesting application I found is using decision tree in geoscience research. By applying C5.0 decision tree algorithm, the author was able to efficiently identify ten distinct mineral groups (olivine, orthopyroxene, clinopyroxene, apatite, amphibole, plagioclase, K-feldspar, zircon, magnetite, biotite) from different igneous rocks.

C5.0 uses the concept of entropy for measuring purity, and is faster and more memory efficient than C4.5. C5.0 gets similar result to C4.5 but output a smaller tree. This echo to the Occam's Razer Theorem professor mentioned in class that for two models providing a similar result, simpler is always better.

Another application is to use decision tree for face detection. It uses decision tree as the last stage of the whole process which determines if a face is present in the image. It has some preprocessing steps, such as finding possible location of the face box. I am surprised that this decision tree algorithm can reach an overall 92% accuracy rate on test data.

In conclusion, decision tree algorithm is simple but yet powerful with appropriate parameters and pruning techniques.

Reference:

https://www.sciencedirect.com/science/article/pii/S0098300415000722

https://www.semanticscholar.org/paper/Detection-of-human-faces-using-decision-trees-Huang-Gutta/8e3ccd490466445790e246cf7e993d450ee354ba