Programming Assignment 1: Decision Trees

Member: Luyao Wang, Di Jin, Yingqi Lin

Individual contributions:

Luyao Wang:

- Discussion about ID 3 algorithm
- Implementation of decision tree and prediction function in python
- Discussion about CART algorithm
- Partial report of part 1 “Implementation”

Di Jin:

- Discussion about ID 3 algorithm
- Discussion about CART algorithm
- Partial report of part 1 “Implementation”
- Partial report of part 2 “Software Familiarization”

Yingqi Lin:

- Discussion about ID 3 algorithm
- Discussion about CART algorithm
- Partial report of part 2 “Software Familiarization”
- Report of part 3 “Applications”

Part 1: Implementation

1. Implementation of Decision Tree and Prediction

(note: in the data.txt, we change one of the original attributes’ name “Favorite Beer” into “FavotiteBeer”)

```
import pdb
import pandas as pd
from math import log
from anytree import Node, RenderTree
from anytree.dotexport import RenderTreeGraph
import fractions
import math

def create_decision_tree_id3(list1, y_col):

    def weight(list1,attribute,value):#return a fraction
        #get the column number of an attribut
        item=find_item(list1,attribute)
        #how many values
        k=1 #loop variable
        count=0 #number of values
        while k<len(list1):
            if list1[k][item]==value:
                count=count+1
            k=k+1
        w=fractions.Fraction(count,len(list1)-1)
        return w
```

```
def possibility(list1,attribute,value,lable_value):#return a fraction
     #get the column number of an attribut
    item=find_item(list1,attribute)
    #how many values and how many "yes"/"No"
    k=1 #loop variable
    count=0 #number of values
    count_lable=0 #number of "yes" or "no"
    while k<len(list1):
        if list1[k][item]==value:
            count=count+1
            if list1[k][-1]==lable_value:
                count_lable=count_lable+1
        k=k+1
    p=fractions.Fraction(count_lable,count)
    return p

def entropy_D(list1):#entropy of root
    p=weight(list1,'Enjoy','Yes')
    entropy= -p*(math.log(p,2))-(1-p)*(math.log(1-p,2))
    return entropy

def entropy_D1(p):# entropy of nonroot
    if p==1 or p==0:
        entropy= 0
    else:
        entropy= -p*(math.log(p,2))-(1-p)*(math.log(1-p,2))
    return entropy
```

```
def entropy_A_D(list1,attribute):#average entropy
    txt = distinct_words(list1,attribute)
    en = [] #entropies of every values in a particular attribute
    for t in txt:
        p=possibility(list1,attribute,t,'Yes')
        en_p=entropy_D1(p)
        w=weight(list1,attribute,t)
        en_w=w*en_p
        en.append(en_w)
    return sum(en)


def get_max_info_gain(list1,y_col): # return the name of attribute
    d = {}
    h = entropy_D(list1)
    for c in list1[0]:
        #get the column number of an attribut
        item=find_item(list1,c)
        if c!=y_col:
            if list1[1][item]!='':
                d[c] = h - entropy_A_D(list1,c)
    if not d:
        pass #d is empty
    else:
        return max(d,key=d.get)
```

```
    def train_decision_tree(node, list1, y_col):
        c = get_max_info_gain(list1, y_col)
        if c!=None:# only have the label: enjoy
            for v in distinct_words(list2,c):
                if v in distinct_words(list2,c) and v not in distinct_words(list1,c):
                    # count the number of "yes"/"no"
                    k=1 #control the loop
                    count_lable_y=0
                    count_lable_n=0
                    while k<len(list1):
                        if list1[k][-1]=='Yes':
                            count_lable_y=count_lable_y+1
                        else:
                            count_lable_n=count_lable_n+1
                        k=k+1
                    if count_lable_y>=count_lable_n:
                        curr_node=Node('%s %s'%(c,v), parent=node)
                        Node('%s %s'%('Enjoy','Yes'), parent=curr_node)
                    else:
                        curr_node=Node('%s %s'%(c,v), parent=node)
                        Node('%s %s'%('Enjoy','No'), parent=curr_node)
                else:
                    if v=='':
                        continue
                    else:
                        p=possibility(list1,c,v,'Yes')
                        curr_node = Node('%s %s'%(c,v),parent=node)
                        # if the attributes are not used up
                        if (len(list1[1])-list1[1].count('')) > 2:
```

```
                        # if the attributes are not used up
                        if (len(list1[1])-list1[1].count('')) > 2:
                            if p == 1:# if the purity is 100%, a leaf node "Yes" is generated
                                Node('Enjoy Yes', parent=curr_node)
                            elif p == 0: #if the purity is 100%, a leaf node "No" is generated
                                Node('Enjoy No', parent=curr_node)
                            else:  #not pure
                                #get the column number of an attribut
                                item=find_item(list1,c)
                                #withsraw a sub list and replace the column of attribute "c" with ''
                                k=1
                                sub_list1=[]
                                sub_list1.append(list1[0])
                                while k<len(list1):
                                    if(v==list1[k][item]):
                                        sub_list1.append(list1[k])
                                        sub_list1[-1][item]= sub_list1[-1][item].replace(sub_list1[-1][item],'')
                                    k=k+1
                                train_decision_tree(curr_node, sub_list1, y_col)
                        #if the attributs run out
                        else:
                            q=1
                            while q<len(list1[1])-1:
                                if list1[1][q]!='':
                                    Node('Enjoy No', parent=curr_node)
                                q=q+1
```

```
    root_node = Node('root')
    train_decision_tree(root_node, list1, y_col)
    return root_node

data=read_data('dt_data.txt')
root_node = create_decision_tree_id3(data, 'Enjoy')
for pre, fill, node in RenderTree(root_node):
    print("%s%s" % (pre, node.name))
RenderTreeGraph(root_node).to_picture("decision_tree_id3.png")
```

```
import copy
def predict(root_node, test, label):
    global list2
    root = copy.deepcopy(root_node)
    root.children[0].name=root.children[0].name.split(' ')
    if root.children[0].name[0]!=label:
        item=find_item(list2,root.children[0].name[0])
        i=0
        while i<len(root.children):
            if i!=0:
                root.children[i].name=root.children[i].name.split(' ')
            #print(i,len(root.children),test[item-1],root.children[i].name[1])
            if test[item-1]==root.children[i].name[1]:
                next_root=root.children[i]
                return predict(next_root,test,label)
            i=i+1
        return root.children[0].name[1]
    else:
        return root.children[0].name[1]
```

```
df=read_data('dt_data.txt')
root_node = create_decision_tree_id3(df, 'Enjoy')
print(predict(root_node,['Moderate','Cheap','Loud','CityCenter','No','No'],'Enjoy'))
print(predict(root_node,['Low','Normal','Quiet','CityCenter','No','No'],'Enjoy'))
```

```
Yes
No
```

2. Data Storage Structure

(1) List

Firstly, we store data in a list. The first line is our attributes and label, the first column is the mark number of each line which will be replaced by ‘ ’ later. Thus, the content of data will start at list[1][1] until list[n][n].In this way, it is convenient for us to assign, modify, add, and delete data in the list. At the same time, we also can index, slice, and divide the list.

```
import re

def remove_special_characters(word):
    regex = re.compile('[^a-zA-Z]')
    return regex.sub('', word)

list2=[] #two-dimensional list

def read_data(filename):
    global list2
    i=0 # total number of rows
    j=0 # total number of columns

    f=open(filename,'r')
    list2=f.read()
    list2=list2.split("\n")
    for i in range(len(list2)):
        list2[i]=list2[i].split()
    for i in range(len(list2)):
        for j in range(len(list2[i])):
            list2[i][j]=remove_special_characters(list2[i][j])
    return list2
```

```
list2=read_data('dt_data.txt')
print (list2)
```

```
[['', 'Occupied', 'Price', 'Music', 'Location', 'VIP', 'FavoriteBeer', 'Enjoy'], [
'', 'High', 'Expensive', 'Loud', 'Talpiot', 'No', 'No', 'No'], ['', 'High', 'Expen
sive', 'Loud', 'CityCenter', 'Yes', 'No', 'Yes'], ['', 'Moderate', 'Normal', 'Quie
t', 'CityCenter', 'No', 'Yes', 'Yes'], ['', 'Moderate', 'Expensive', 'Quiet', 'Ger
manColony', 'No', 'No', 'No'], ['', 'Moderate', 'Expensive', 'Quiet', 'GermanColon
y', 'Yes', 'Yes', 'Yes'], ['', 'Moderate', 'Normal', 'Quiet', 'EinKarem', 'No', 'N
o', 'Yes'], ['', 'Low', 'Normal', 'Quiet', 'EinKarem', 'No', 'No', 'No'], ['', 'Mo
derate', 'Cheap', 'Loud', 'MahaneYehuda', 'No', 'No', 'Yes'], ['', 'High', 'Expens
ive', 'Loud', 'CityCenter', 'Yes', 'Yes', 'Yes'], ['', 'Low', 'Cheap', 'Quiet', 'C
ityCenter', 'No', 'No', 'No'], ['', 'Moderate', 'Cheap', 'Loud', 'Talpiot', 'No',
'Yes', 'No'], ['', 'Low', 'Cheap', 'Quiet', 'Talpiot', 'Yes', 'Yes', 'No'], ['', '
Moderate', 'Expensive', 'Quiet', 'MahaneYehuda', 'No', 'Yes', 'Yes'], ['', 'High',
'Normal', 'Loud', 'MahaneYehuda', 'Yes', 'Yes', 'Yes'], ['', 'Moderate', 'Normal',
'Loud', 'EinKarem', 'No', 'Yes', 'Yes'], ['', 'High', 'Normal', 'Quiet', 'GermanCo
lony', 'No', 'No', 'No'], ['', 'High', 'Cheap', 'Loud', 'CityCenter', 'No', 'Yes',
'Yes'], ['', 'Low', 'Normal', 'Quiet', 'CityCenter', 'No', 'No', 'No'], ['', 'Low'
, 'Expensive', 'Loud', 'MahaneYehuda', 'No', 'No', 'No'], ['', 'Moderate', 'Normal
', 'Quiet', 'Talpiot', 'No', 'No', 'Yes'], ['', 'Low', 'Normal', 'Quiet', 'CityCen
ter', 'No', 'No', 'Yes'], ['', 'Low', 'Cheap', 'Loud', 'EinKarem', 'Yes', 'Yes', '
Yes']]
```

(2) Node

Secondly, we use nodes to store the name of the current and the parent of the current node. And our function ”create_decision_tree_id3(dataset, label)” returns a root node. Thus, we can build or withdraw a tree only if we know the root node.

```
    else:
        p=possibility(list1,c,v,'Yes')
        curr_node = Node('%s %s'%(c,v),parent=node)
        # if the attributes are not used up
        if (len(list1[1])-list1[1].count('')) > 2:
            if p == 1:# if the purity is 100%, a leaf node "Yes" is generated
                Node('Enjoy Yes', parent=curr_node)
```

```
root_node = create_decision_tree_id3(data, 'Enjoy')
for pre, fill, node in RenderTree(root_node):
    print("%s%s" % (pre, node.name))
```

3. Code-level Improvement

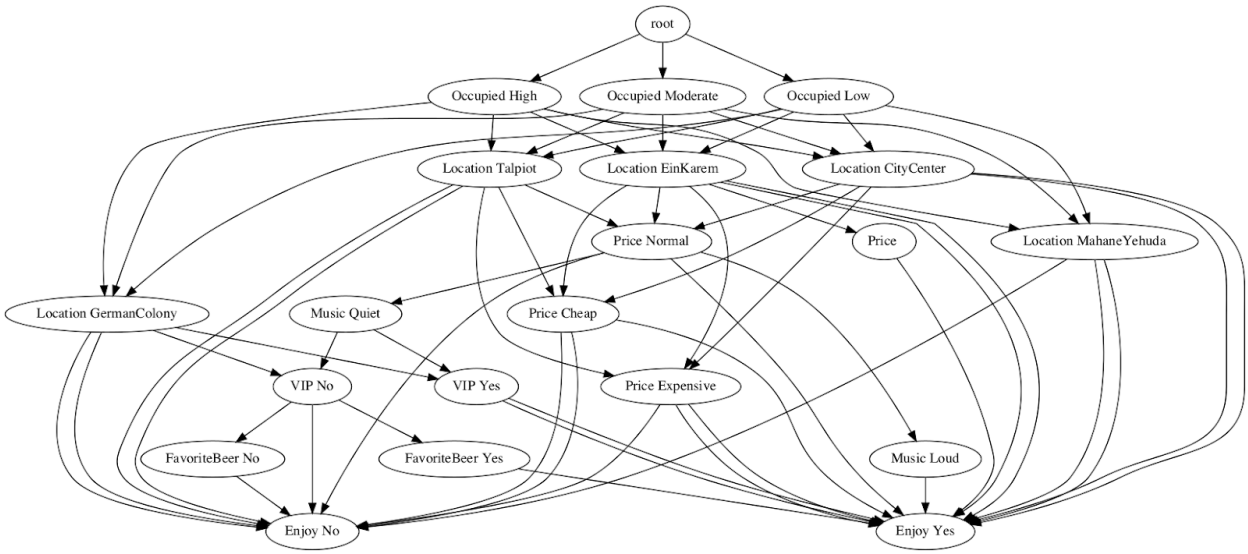
(1) reduction of repeated code

We withdraw some code as independent functions outside so that we only need to call the function's names when we need them. Thus, the repetition of code is reduced a lot. For example, the function find_item(list1,attribute) and the function distinct_words(list1,attribute).

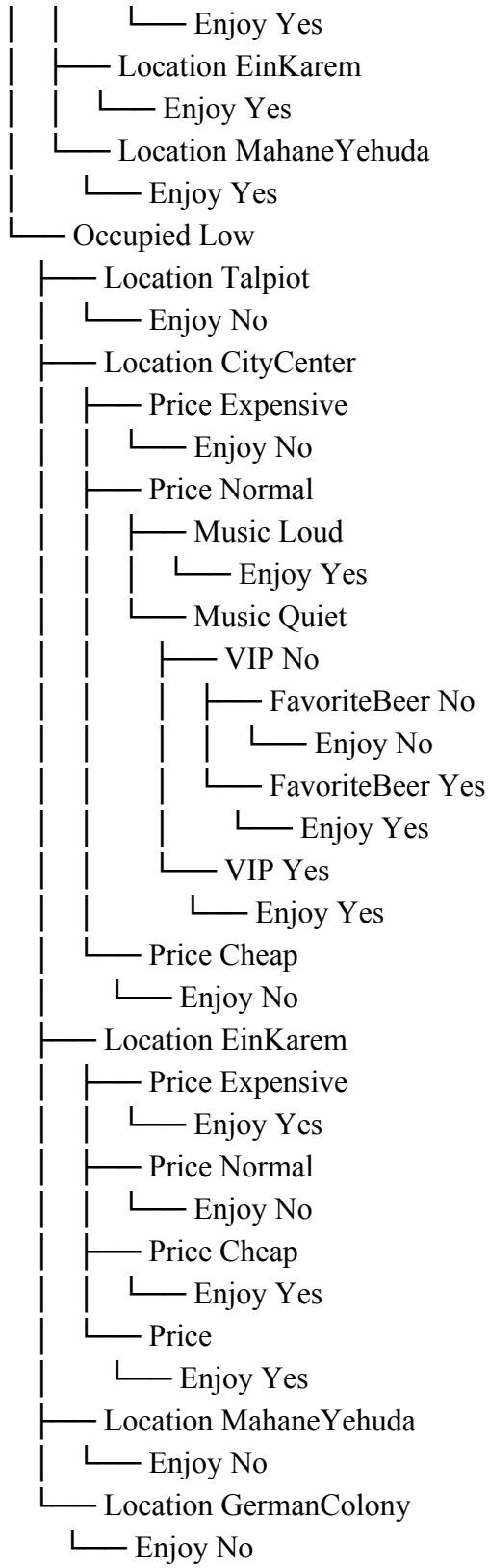
```
#get the column number of an attribut
def find_item(list1,attribute):
    item=0 #对应的列数
    while item < len(list1[0]):
        if attribute==list1[0][item]:
            break
        else:
            item=item+1
    return item

#get a list only contains the unique values of an attribute
def distinct_words(list1,attribute):
        #find the column number of this attribute
        item=find_item(list1,attribute)
        #change the column to the One-dimensional list
        k=1
        text=[]
        while k<len(list1):
            text.append(list1[k][item])
            k=k+1
        #put the unique values into text_final[]
        text_final = []
        for t in text:
            if t not in text_final:
                text_final.append(t)
        return text_final
```

(2) Visualization of Decision Tree

In the code level optimization, we reduce the repetition rate of the function and improve coding efficiency. Unifying all variable names and adding comments make the code simple and easy to read. In the visualization of the decision tree, we import the Render tree, which is easy to see the generation of a decision tree.



root
- Occupied High
  - Location Talpiot
    - Enjoy No
  - Location CityCenter
    - Enjoy Yes
  - Location GermanColony
    - Enjoy No
  - Location EinKarem
    - Enjoy Yes
  - Location MahaneYehuda
    - Enjoy Yes
- Occupied Moderate
  - Location Talpiot
    - Price Expensive
      - Enjoy Yes
    - Price Normal
      - Enjoy Yes
    - Price Cheap
      - Enjoy No
  - Location CityCenter
    - Enjoy Yes
  - Location GermanColony
    - VIP No
      - Enjoy No
    - VIP Yes


Enjoy Yes
Location EinKarem
Enjoy Yes
Location MahaneYehuda
Enjoy Yes
Occupied Low
Location Talpiot
Enjoy No
Location CityCenter
Price Expensive
Enjoy No
Price Normal
Music Loud
Enjoy Yes
Music Quiet
VIP No
FavoriteBeer No
Enjoy No
FavoriteBeer Yes
Enjoy Yes
VIP Yes
Enjoy Yes
Price Cheap
Enjoy No
Location EinKarem
Price Expensive
Enjoy Yes
Price Normal
Enjoy No
Price Cheap
Enjoy Yes
Price
Enjoy Yes
Location MahaneYehuda
Enjoy No
Location GermanColony
Enjoy No

(3) Simple and Readable Coding Style

Unifying all variable names and adding clear comments so that the code will be simple and easy to read.

4. Challenges

(1) Lack of Knowledge of Data Storage Structure “Dictionary”

Firstly, we are not familiar with the data structure “Dictionary”. So, we don’t use the dictionary, which is a more efficient storage mode. Instead, we use the list storage mode. In this way, it increases the time we spend searching and inserting data when the number of data increases. If we deal with a large amount of data, the processing speed of our code will be very slow.

(2) Not familiar with Iteration

Secondly, we are not familiar with the idea of iteration in our algorithm, which leads us to calculate the root node, but can not calculate the next node. Later, we solved this problem by learning the idea of iteration.

(3) How to Resolve Data Loss

Thirdly, our algorithm can't deal with the situation that the data of the value of an attribute in the dataset is missing. In this case, if we want to predict the result of a value that the dataset can not search, the program will report an error. Because the database sample is not large enough, and this relevant data is missing. Later, we calculate the number of labels of the value’s parent node and select the bigger one as the label of the value. If the number of labels of the value’s parent node is equal, we can return yes or no randomly.

(4) Flaws of ID3

Besides, we find that the ID3 algorithm itself has some defects. Firstly, ID3 does not consider continuous features, such as length and density. Secondly, ID3 calculates the largest information gain to establish nodes of decision-trees. However, if a value has more data information, this value tends to have larger information gain than other values in the same attribute. Finally, ID3 does not consider the problem of overfitting.

Part2: Software Familiarization

1. How to use the built-in function DecisionTreeClassifier( )

The library function that offers a good implementation of the decision tree learning algorithm called CART(Classification and Regression Tree) which uses the Gini index to measure the divergences between the probability distributions of the target attribute’s values. Also, the Gini index represents the rate of impurity and it is defined as Gini Index = $1-\sum [p\ (i/t)]$

Here are some steps to use the CART function to implement the decision tree.

Step1: Importing some required libraries.

```
from sklearn.datasets import load_iris
```

from sklearn.tree import DecisionTreeClassifier
from sklearn.tree.export import export_text

Step2: Loading Data

Iris = load_iris( )

Step3: Feature Selection

Now, we need to divide given columns into two types of variables dependent(or target variable) as the label of Enjoy in our assignment and independent variable(or feature variables).

Step4: Splitting Data

Then the data should be divided into a training set and a test set using function train_test_split( ).

Step5: Building Decision Tree Model

clf = tree.DecisionTreeClassifier( )

Step6: Evaluating Model

We don't know how accurately the classifier or model can predict the type of cultivars so we should compute by comparing actual test set values and predicted values.

Step7: Visualizing Decision Trees

tree.plot_tree(clf.fit(iris.data, iris.target))

Or we can install graphviz and pydotplus to visualize our decision tree.

Some changes of parameters can optimize the performance of the Decision Tree like criterion, splitter, and max_depth. For the criterion, we can choose the "Gini" for the Gini impurity or "entropy" for the information gain and the default is "Gini" for the Gini impurity. For the splitter, there are two ways that are "best" to choose the best split and "random" to choose the best random split and the default is "best" which is adapted to a small sample. For the max_depth, the default is none. When the sample is large, it is better for us to set the max_depth. If none, then nodes are added until all the leaves contain less than min_samples_split samples. A high max_depth value will cause overfitting.

2. Contrast and Improve:

Firstly, referring to the CART algorithm, we can use the Gini method to process continuous variables as discrete variables. Each split point is the mean value of two adjacent values. Coping with multiple discrete values, they will be separated into two classes. For example, in our algorithm, the three values, expensive, normal and low are randomly divided into two parts. And then, we calculate their Gini values and select the way of dividing with the lowest Gini value.

Secondly, the problem of overfitting would happen in the ID3 algorithm. However, in the CART algorithm, We can prune some extra leaves which don't work for the classifier. At the same time, we can achieve a low misclassification and high-efficiency decision tree. CART

algorithm picks up the cost-complexity pruning algorithm which is one of the post-pruning ways. For the cost-complexity pruning algorithm, we need to calculate the index of alpha and it is defined as $\alpha=(R(t)-R(Tt))/|N\text{Ti}|-1$. The $R(t)-R(Tt)$ represents misclassification and the $|N\text{Ti}|-1$ represents the amount of decreased leaves. So we choose the minimum of the index of alpha which is the node we should prune. When pruning, the selected node's children should be dropped. If $\alpha=0$ then the biggest tree will be chosen. As α approaches infinity, the tree of size 1, a single root node, will be selected.

Part3: Applications

In real life, we often use decision-trees to decide our life. For example, we always use decision-trees to decide what activities to schedule for our weekend. What kind of activities to take may depend on some factors. The first one is whether we are willing to go out with friends or spend the weekend alone and the second one is the weather of the weekend. If we assume that these two factors affect our decision, if it's sunny and our friends can participate together, we might want to play football. If it is rainy, we might go to the cinema together. If a friend is unable to participate in something, we might read books and play video games regardless of the weather.

References:

https://machinelearningmastery.com/classification-and-regression-trees-for-machine-learning/

https://scikit-learn.org/dev/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier

https://scikit-learn.org/stable/modules/tree.html

https://www.datacamp.com/community/tutorials/decision-tree-classification-python