[Assignment report #2]

Classification by Decision Tree Algorithm

2012023871
경영학과
강민철

1) Summary of Algorithm

Classification algorithm trains a classifier model from train data which has both attributes and label. Then it classifies a test data which only contains attributes to each label based on the trained algorithm. The train of decision tree algorithm starts with finding the best attribute. In this case, the best attribute means it draws the best information gain among given attributes. Then it splits the train set into left and right node with respect to the best attribute.

It repeats this single process 'recursively' until some conditions are achieved. One is that there are no data to split and the other is the information gain you get after the split is minus.

2) Detailed Description of Codes

[Library Import]

```
from optparse import OptionParser
from math import log
```

- 'Optionparser' enables a .py file to take arguments on the bash level.
- 'chain and combinations' are functions needed to process set operator such as generating subset.
- 'log' is used to calculate entropy value

[Main function]

```
if __name__ == "__main__":
    # optionParser setting
    optparser = OptionParser()

    optparser.add_option('-t', '--train', dest='train',
                         help='train data file name',
                         default='dt_train.txt')
    optparser.add_option('-v', '--test', dest='test',
                         help='test data file name',
                         default='dt_test.txt')
    optparser.add_option('-r', '--result', dest='result',
                         help='result file name', default='dt_result.txt')

    (options, args) = optparser.parse_args()

    #Read train, test file
    data = FileReader(options.train)
    test_a = FileReader(options.test)
    header = data[0]        #Extract header for output's first row
    test = test_a[1:]       #Omit header row
    my_data = data[1:]      #Omit header row

    #Recursively find the best attribute
    tree = RecursiveBuild(my_data)

    #Make a prediction based on trained decision tree
    result = ClassifyAll(test, tree)

    #Make output file with given name
    out_file = open(options.result, 'w')

    #Write predictions on the output file
    Result = WriteFile(result, out_file, header)
    out_file.close()
```

First few lines are for setting the OptionParser. Specifically, 1st option takes the name of train data from users and store it in the data instance, 2nd option takes the name of test data file name while 3rd option takes the file name of result file.

Then it reads each dataset with FileReader function. It extracts header of train data for the output's first row. (The assignment guideline dictates so) As you don't need the attribute name, it omits the header row of train and test data. It proceeds to recursively find the best split using train data. After you build a trained tree, you can make a prediction based on it. The ClassifyAll function does this prediction and returns a result which has same format of train data (including label column). WriteFile actually writes the result into the output file which takes the options.result as its name. Then ends the whole code with closing the output file.

[FileReader function]

```
def FileReader(file_path):

    #Read each row
    #in accordance with the given format
    df = []
    file_iter = open(file_path, 'rU')
    for row in file_iter:
        row = row.strip().rstrip('\t')
        tup = row.split('\t')
        df.append(tup)
    return df
```

This function opens the train and test data file by iterating each row of the data. Watch that the delimeter is '₩t'.

[RecursiveBuild function -1]

```
def RecursiveBuild(df):

    #Local function for entropy calculation
    def entropy(df):
        results = CountCheck(df)
        ent = 0.0
        for label in results:
            p = float(results[label]) / len(df)
            ent = ent - p * log(p) / log(2)
        return ent

    #Condition for recursion end
    if len(df) == 0:
        return TrNode()

    current_score = entropy(df)

    top_value = 0.0     #Information gain
    top_att = None      #Best attribute
    top_splits = None   #Best splitted two sets


    column_count = len(df[0]) - 1
```

First it starts with defining local function which calculates entropy of a given data. Then it first defines the condition for recursion end. (if len(df) == 0) means that there are no rows left to split. Current_score contains the entropy of original train data at this time. Then it initializes the instance for best information gain, best

attribute, and the divided two sets with respect to the attribute. Colum_count is the index where splits the attributes and label of train data.

[RecursiveBuild function -2]

```
#loop through every attributes except label
for attr in xrange(0, column_count):

    column_att_values = {}
    for row in df:
        column_att_values[row[attr]] = 1

    for att_value in column_att_values.keys():
        #Split based on certain value
        #Calculate information gain
        (left, right) = SplitSet(df, attr, att_value)

        p = float(len(left)) / len(df)
        gain = current_score - p * entropy(left) - (1 - p) * entropy(right)

        #Updates top_value and so on
        if gain > top_value and len(left) > 0 and len(right) > 0:
            top_value = gain
            top_att = (attr, att_value)
            top_splits = (left, right)
#Repeat until the information gain get minus
if top_value > 0:
    Leftbranch = RecursiveBuild(top_splits[0])
    Rightbranch = RecursiveBuild(top_splits[1])

    #Recursively build decision tree
    #using pre-defined node object
    return TrNode(attr=top_att[0], att_value=top_att[1],
                  left=Leftbranch, right=Rightbranch)
else:
    #When it reach leaf, count the number
    return TrNode(results=CountCheck(df))
```

Then it loops through every given attributes of train data, split the data based on certain value of each attributes and calculates the information gain. If the information gain of certain split is greater than previous top_value(the best information gain so far, it updates the top_value, top_att, and top_splits. It recursively repeats this process until the information gain gets minus. It builds the tree by connecting the nodes started from the root node. Watch "Leftbranch = RecursiveBuild() & Rightbranch = RecursiveBuild()". When it reaches the leaf node, it write the number of each element of label on the node object's results which acts

as a exit condition (look at the RecursiveBuild part 1). The count is processed by CountCheck function.

[SplitSet function]

```
# Split based on whether each row has
# given column's certain value
def SplitSet(df, column, att_value):
    left = [row for row in df if row[column] == att_value ]
    right = [row for row in df if not row[column] == att_value]

    return (left, right)
```

This function appears in the RecursiveBuild function. The role of this function is to split the given train data by the given column and attribute value.

[CountCheck function]

```
#Count the number of leaf element
def CountCheck(tups):
    res = {}
    for t in tups:
        label = t[len(t) - 1]
        if label not in res:
            res[label] = 0
        res[label] += 1
    return res
```

This function counts the number of leaf element. First it casts an empty dictionary. Then when it finds the label element for the first time, it makes the keys in res. For the length of train data, it continues to count the number of elements.

[ClassifyAll function]

```
def ClassifyAll(test_data, tree):

    #local function to classify each row
    def classify(row, tree):
        #When leaf node
        if tree.results != None:
            return tree.results
        else:
            #Follow from the root node to leaf
            v = row[tree.attr]
            branch = None

            if v == tree.att_value:
                branch = tree.left
            else:
                branch = tree.right

            #Repeat until it gets to the leaf
            return classify(row, branch)
    #Apply classify function to the whole test data
    for i in xrange(len(test_data)):
        #Make a prediction for each row
        classified = classify(test_data[i], tree).keys()

        #Then append it to final column
        test_data[i].append(classified[0])

    return test_data
```

This function makes predictions based on the trained decision tree. First it makes local function to classify each row of test data. The local function follows from the root node to an arbitrary leaf node based on the observations of each rows. As the condition of reaching to the leaf node varies for each rows, it used recursion again (Watch return classify()).

The ClassifyAll function applies classify function to the whole test data and append the classified result of each row to the final column of test data.

[CheckAccuracy function]

```
#Unused in current version
#For calculating accuracy
def CheckAccuracy(result, Test_real):
    end_point = len(result[0])-1

    #Count correct prediction
    cnt = 0
    for i in xrange(len(result)):
        if result[i][end_point] == Test_real[i][end_point]:
            cnt +=1
    accuracy = float(cnt)/len(result)

    return accuracy
```

Although this function is not used in current version, I used it for improving the previous code. It compares the prediction set and actual answer and add 1 to the cnt when it made a correct answer.

[WriteFile function]

```
def WriteFile(test_data, f, header):

    attr_len = len(test_data)
    row_len = len(test_data[0])

    #Write first rows: header
    first = ""
    for k in xrange(len(header)):
        first += "%s\t" % (header[k])
    first += "\n"
    f.write(first)

    #Unlist the array to meet
    #guided output format
    for i in xrange(attr_len):
        row = ""
        for j in xrange(row_len):
            row += "%s\t" % (test_data[i][j])
        row += "\n"

        f.write(row)

    return f
```

First, it makes a 'first' instance to write the header to the output file. Then it loops through the result file and write the result to the output file in accordance with the output format which the guideline dictated.

3) Instructions for compiling the source codes

[Step 1]: copy the repository and change directory to ~/project_apriori/

```
git clone http://hconnect.hanyang.ac.kr/2017_ITE4005_10065/2017_ITE4005_2012023871.git
```

- Copy my gitlab repository by using *git clone* or download the *zip file* from the website. [http://hconnect.hanyang.ac.kr/2017_ITE4005_10065/2017_ITE4005_2012023871]

```
cd DS_git/2017_ITE4005_2012023871/Programming_Assignment_1/project_apriori/
```

- go the the project_dt directory by 'cd' command.

[Step 2] : Print help message about arguments

```
python dt.py -h
```

- type *$python dt.py –h* on command line then it returns the help message of dt.py. (see below)

```
Usage: dt.py [options]

Options:
  -h, --help            show this help message and exit
  -t TRAIN, --train=TRAIN
                        train data file name
  -v TEST, --test=TEST  test data file name
  -r RESULT, --result=RESULT
                        result file name
```

-t(--train): name of train data

-v(--test): name of test data

-r(--result): name of output data

[Caution] : the train data and test data should be in the same directory of dt.py

[Step 3] : Actually run the code by python (2.7 ONLY)

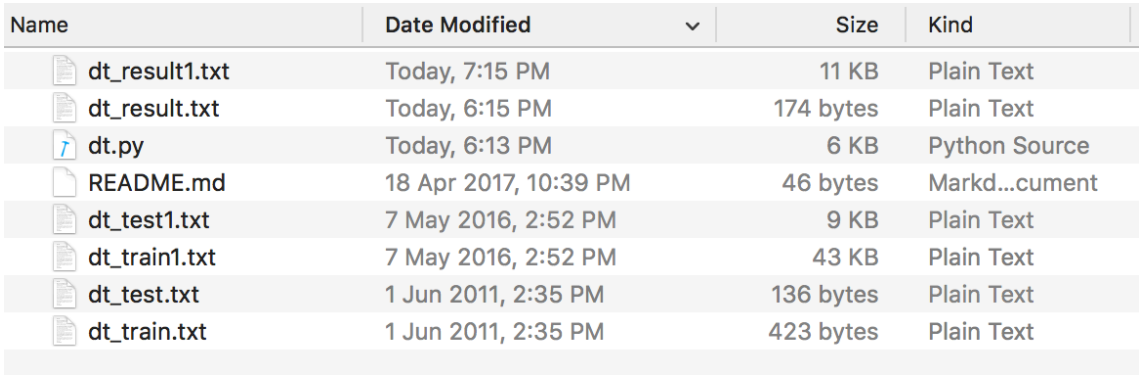
```
python dt.py --train dt_train1.txt --test dt_test1.txt --result dt_result1.txt
```

- run

  $python dt.py --train dt_train1.txt --test dt_test1.txt --result dt_result1.txt

  on terminal or command line

  [Caution] python should be 2.7 # python3 will cause error



Name	Date Modified	Size	Kind
dt_result1.txt	Today, 7:15 PM	11 KB	Plain Text
dt_result.txt	Today, 6:15 PM	174 bytes	Plain Text
dt.py	Today, 6:13 PM	6 KB	Python Source
README.md	18 Apr 2017, 10:39 PM	46 bytes	Markd...cument
dt_test1.txt	7 May 2016, 2:52 PM	9 KB	Plain Text
dt_train1.txt	7 May 2016, 2:52 PM	43 KB	Plain Text
dt_test.txt	1 Jun 2011, 2:35 PM	136 bytes	Plain Text
dt_train.txt	1 Jun 2011, 2:35 PM	423 bytes	Plain Text

- Then you will see the "dt_result1.txt" file has been made in the same directory.

It looks like below:

buying	maint	doors	persons	lug_boot	safety	car_evaluation
med	vhigh	2	4	med	med	unacc
low	high	4	4	small	low	unacc
high	vhigh	4	4	med	med	unacc
high	vhigh	4	more	big	low	unacc
low	high	3	more	med	low	unacc
med	high	2	more	small	high	unacc
vhigh	low	3	2	med	high	unacc
med	high	2	4	small	low	unacc
med	low	5more	4	small	med	acc
med	low	5more	2	big	med	unacc
med	low	4	more	big	high	vgood
low	low	4	2	big	high	unacc
low	low	3	more	med	low	unacc
high	med	2	2	big	high	unacc
high	low	4	more	small	low	unacc
med	vhigh	3	4	med	med	unacc
low	low	3	more	small	high	good
vhigh	med	2	more	med	med	unacc
vhigh	low	4	more	big	high	acc
vhigh	low	2	2	small	high	unacc
high	high	5more	2	big	med	unacc
high	med	5more	2	med	low	unacc
med	med	3	2	big	high	unacc
low	vhigh	5more	2	small	low	unacc
vhigh	vhigh	3	more	small	high	unacc
low	high	2	more	big	med	acc
vhigh	vhigh	5more	more	small	high	unacc
high	low	5more	4	small	med	unacc
low	med	4	2	big	low	unacc