CSC <535/635> Data Mining

Assignment <2> Report

Submitted to:
Dr. Jamil Saquer

Author(s):
<Tran Tien Dung >

Assignment 2 Report

Introduction

In the assignment 2, we learn about the decision tree and how to implement ID3 algorithm. Basically, a decision tree is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance, event, outcomes, resource costs and utility [1]. Decision tree can be generated from a wide range of methods and one of them is known as Iterative Dichotomiser 3 (ID3), which repeatedly divides features into two or more groups at each step. Most generally ID3 is used for classification problems with a nominal number of features.

In my part 2 of this assignment, I adopt the data set from https://www.kaggle.com about the Titanic disaster [2] that was created to predict which passengers survived over the Titanic shipwreck. To deal with this issue, I apply the ID3 algorithm that has been done in the previous part and also propose some other approaches that is aimed to increase the accuracy rate of the model. About the Titanic data set, it has two different files "train.csv" and "test.csv". Because the "test.csv" file does not have the ground truth, so I split the "train.csv" which has information of 891 passengers into two different sections. First section including 650 passengers is used for training purpose. Second section which contains the rest of the data is used to evaluate the model. The first 4 rows of dataset are as follow:

	PassengerId	Survived	Pclass	...	Fare	Cabin	Embarked
0	1	0	3	...	7.2500	NaN	S
1	2	1	1	...	71.2833	C85	C
2	3	1	3	...	7.9250	NaN	S
3	4	1	1	...	53.1000	C123	S

With more than 10 attributes and the job is to determine if the passengers are survived or not, the dataset is an ideal object for implementing and checking the efficiency of ID3 algorithm. Notice that some attributes including ['PassengerId', 'Name', 'Ticket', 'Cabin', 'SibSp] are not suitable for classification, so I drop it out of the list.

Background

ID3 is an algorithm invented by Ross Quinlan [3]. It uses a top-down greedy approach to build a decision tree. To be more extent, the top-down greedy approach means that a tree is created from the top and then at each iteration the best feature is used to form a node. The heart of this algorithm is that it takes advantage of Entropy to calculate the information gain of each feature and then the feature with the highest information gain will be defined as the best. At first, Entropy is the measure of disorder and the Entropy of a dataset is the measure of disorder in the target feature of the dataset. Entropy of dataset D is calculated as:

$$entropy(D) = -\sum_{j=1}^{|C|} \Pr(c_j) \log_2 \Pr(c_j)$$

$$\sum^{|C|} \Pr(c_j) = 1,$$

Where pr(c_j) is the probability of class C_J. Then the information gain for selecting feature A_i is:

$$gain(D, A_i) = entropy(D) - entropy_{A_i}(D)$$

Implementation

In my main program for generating the decision tree, I used two different classes and one function. The first class named "TreeNode" is to define each node in the tree.

```
class TreeNode(object):

    def __init__(self, ids = None, children = [], entropy = 0, depth = 0):
        self.ids = ids           # index of data in this node
        self.entropy = entropy   # entropy, will fill later
        self.depth = depth       # distance to root node
        self.split_attribute = None # which attribute is chosen, it non-leaf
        self.children = children # list of its child nodes
        self.order = None        # order of values of split_attribute in children
        self.label = None        # label of node if it is a leaf
        self.defautl_label = None # a defualt label for unexpected an missing attribute
```

The second class is the main ID3 tree which will calculate the information gain of each attribute and then build up the tree.

```
class DecisionTreeID3(object):

    #initilize Data
    def __init__(self, max_depth= 10, min_samples_split = 2, min_gain = 1e-3):
        self.root = None
        self.max_depth = max_depth     # limitation depth of the tree
        self.Ntrain = 0
        self.min_gain = min_gain      # minimum threshold for the information gain
        self.min_samples_split = min_samples_split
```

And finally, a function to calculate Entropy is set up outside of these two classes. I employed Numpy, Panda and Math in order to make the job more convenient. Besides, I also add in the tree some more variables to prevent overfitting and reach to higher accuracy rate, which will be presented in the following section.

Improvement Propose

About the decision tree algorithm, generally if we keep on dividing an impurity node, we can eventually get a tree in which all the points in training set are predicted correctly. However, the tree can be too complicated with a large number of leaf nodes and hence possibly cause overfitting. In order to avoid this, I used some other conditions to stop the tree as below:

- Min_samples_split: if the node has a number of elements that is smaller than a certain threshold, the algorithm will stop and the class for leaf node will be determine by the most common class in node.
- Max_depth: Stop the tree if the distance from the node to root node get to a threshold. This helps to decrease the complexity of algorithm and avoid overfitting
- Min_gain: if the information gain is smaller than Min_gain, it means that separating this node does not reduce much Entropy and the algorithm can stop.

About the dataset, it is clear that data plays an important role on the performance of algorithm. Similar to the problem of part 1, the raw version of Titanic data set contains a lot of unwanted and missing values. For example, in the attribute [“Age”], some of the values are not presented or in the attribute[“cabin”], it includes a lot of unexpected characters. To handle it, I presented some techniques to process the data before putting it into model as following:

- Creating a new column “Has_cabin” form “Cabin”
- Creating a new column “FamilySize” as a combination of “SibSp” and “Parch”
- Creating a new column “ISAlone” from Familysize
- Mapping “Sex”: [Female, Male] to [0, 1]
- Mapping “Fare”: notice that the fare is a wide range of different float numbers. To make it suitable for ID3, we group it into three different set [0, 1, 2,3] corresponding to [(x <= 7.91), (7.91 < x < 14.454), (14.454 < x < 31), (> 31)].
- Mapping “Emparked”: [S, C, Q] to [0, 1, 2]
- Mapping “Age”: Similar to “Fare”, mapping to 4 groups [0, 1, 2, 3, 4]

In order to make the perfect mapping, we first need to fulfil the missing data. In my program, I used the most frequent values, median and mean to do this task. For example:

- data['Embarked'] = data['Embarked'].fillna('S') // S is the most frequent values
- data['Fare'] = data['Fare'].fillna(data['Fare'].median())
- data.loc[np.isnan(data['Age']), 'Age'] = age_avg

Finally, the data after being processed is:

	Survived	Pclass	Sex	Age	...	Embarked	Has_Cabin	FamilySize	IsAlone
0	0	3	1	1	...	0	0	2	0
1	1	1	0	2	...	1	1	2	0
2	1	3	0	1	...	0	0	1	1
3	1	1	0	2	...	0	1	2	0

Some of the attributes was dropped out as it is not necessary and replaced by new creation. The data now is all numbers that is most suitable for decision tree and potentially helps to increase the accuracy rate.

Experimental Setup and Results

The program is done in Spider IDE, which can be found in Anaconda. Firstly, to know the optimal values for the parameters of ID3, I consider doing experiment in the two data sets.

- Without using parameters and processing data: accuracy rate = 0.7925
- Keep the value min_samples_split = 2 and min_gain = 1e-6, and change the values of Max_depth:

Table 1: The importance of Max_depth in ID3

	1	2	3	4	5	6	7
Raw Data	0.7883	0.7883	0.8049	0.7925	0.7925	0.7925	0.7925
Processed Data	0.7883	0.7883	0.8174	0.8215	0.8215	0.8215	0.8174

- Keep the Max_dept = 5, min_samples_split = 2 and change the values of min_gain:

-

Table 2: The importance of Min_gain

	1e-1	1e-2	1e-3	1e-4	1e-5	1e-6	1e-7
Raw Data	0.7883	0.7925	0.7925	0.7925	0.7925	0.7925	0.7925
Processed Data	0.7883	0.8215	0.8215	0.8215	0.8215	0.8174	0.8215

Since the dataset is not too big, the min_samples_split does not contribute much to the accuracy rate. Therefore, I fix its values as 2.

From table 1 and table 2, we conclude that the best accuracy rate over the test set is at 82.15% with the max_depth = 5, min_gain = 4 and the sample_splits = 2 which means 3% higher than the original result (79.25%). It also proved that pre-processing data before fitting in the ID3 algorithm gives a significant higher result than using the raw data.

Conclusion

In this assignment, I have learned how to implement a decision tree using ID3 algorithm. Specially, I find out that handling unexpected data plays an important role on improving the overall accuracy rate. Moreover, overfitting is one of the most common problem in decision tree and it can be overcome by setting up stopping points for the algorithm or using pruning technique.

References

[1] https://en.wikipedia.org/wiki/Decision_tree

[2] https://www.kaggle.com/c/titanic/data?select=train.csv

[3] Quinlan, J. R. 1986. Induction of Decision Trees. Mach. Learn. 1, 1 (Mar. 1986), 81–106

Code

```
"""
Program: ID3.py
Programmed By: Tran Tien Dung
Description: python file for assignment 2
"""


#---------------------------------Imports-------------------------------------
import pandas as pd
import math
import numpy as np

#---------------------------------Classes/Functions---------------------------

class TreeNode(object):

    def __init__(self, ids = None, children = [], entropy = 0, depth = 0):
        self.ids = ids           # index of data in this node
        self.entropy = entropy   # entropy, will fill later
        self.depth = depth       # distance to root node
        self.split_attribute = None # which attribute is chosen, it non-leaf
        self.children = children # list of its child nodes
        self.order = None       # order of values of split_attribute in children
        self.label = None       # label of node if it is a leaf
        self.defautl_label = None # a defualt label for unexpected an missing attribute

    def set_properties(self, split_attribute, order):
        self.split_attribute = split_attribute
        self.order = order

    def set_label(self, label):
        self.label = label

    def set_default_label(self, label):
        self.defautl_label = label;

#caculate entropy
def entropy(freq):

    if int(freq[0]) == 0 or int(freq[1]) == 0: return 0
    prob_0 = int(freq[0])/ (int(freq[0]) + int(freq[1]))
    temp = - prob_0*math.log(prob_0) - (1-prob_0)*math.log(1-prob_0)
    return temp

class DecisionTreeID3(object):

    #initilize Data
    def __init__(self, max_depth= 10, min_samples_split = 2, min_gain = 1e-3):
```

```
        self.root = None
        self.max_depth = max_depth     # limitation depth of the tree
        self.Ntrain = 0
        self.min_gain = min_gain      # minimum threshold for the information gain
        self.min_samples_split = min_samples_split
    #Fit data to train
    def fit(self, data, target):

        self.Ntrain = data.count()[0]
        self.data = data
        self.attributes = list(data)
        self.target = target
        self.labels = target.unique()

        ids = range(self.Ntrain)
        self.root = TreeNode(ids = ids, entropy = self._entropy(ids), depth = 0) # create
a root of tree

        queue = [self.root]
        while queue:
            node = queue.pop()
            if node.depth < self.max_depth or node.entropy < self.min_gain:
                node.children = self._split(node)
                if not node.children: #check if the node is leaf node
                    self._set_label(node)

                queue += node.children
            else:
                self._set_label(node)

    # take the default_label that is the most common class
    def default_label(self,ids):

        if len(ids) == 0: return 0

        count_y = 0
        count_n = 0

        for element in ids:

             if self.target[element] == "yes" or self.target[element] == True:
                 count_y += 1
             else:
                 count_n += 1

        if count_y > count_n:
            check = True
        else:
            check = False

        return  check

    #caculate entropy for each atribute
    def _entropy(self, ids):

        if len(ids) == 0: return 0

        count_y = 0
        count_n = 0
        freq = []
        for element in ids:
```

```
            if self.target[element] == "yes" or self.target[element] == True or
self.target[element] == 1:
                count_y += 1
            else:
                count_n += 1

        freq.append(count_y)
        freq.append(count_n)
        return entropy(freq)


    def _set_label(self, node):

        node.set_label(self.target[node.ids].mode()[0]) # most frequent label


    #building a ID3 Tree
    def _split(self, node):
        ids = node.ids
        best_gain = 0
        best_splits = []
        best_attribute = None
        order = None
        sub_data = self.data.iloc[ids, :]

        for i, att in enumerate(self.attributes):
            values = self.data.iloc[ids, i].unique().tolist()

            splits = []
            for val in values:
                sub_ids = sub_data.index[sub_data[att] == val].tolist()
                splits.append(sub_ids)

            # don't split if a node has too small number of points
            if min(map(len, splits)) < self.min_samples_split: continue

            # information gain
            H_S= 0
            for split in splits:
                H_S += len(split)*self._entropy(split)/len(ids)

            gain = node.entropy - H_S
            if gain < self.min_gain: continue # stop if small gain
            if gain > best_gain:
                best_gain = gain
                best_splits = splits
                best_attribute = att
                order = values

        node.set_properties(best_attribute, order)
        node.set_default_label(self.default_label(ids))
        child_nodes = [TreeNode(ids = split,
                    entropy = self._entropy(split), depth = node.depth + 1) for split in
best_splits]


        return child_nodes

    #Classify new entry
    def classify(self, new_data,n):
```

```
        labels = []
        for i in range(n):
            x = new_data.iloc[i, :]
            # start from root and recursively travel if not meet a leaf
            node = self.root
            while node.children:

                attribute = x[node.split_attribute]
                if attribute not in node.order:

                        labels.append(node.defautl_label)
                        break
                else:
                        node = node.children[node.order.index(x[node.split_attribute])]
            if node.label == None:
                continue
            else:
                labels.append(node.label)

        return labels
#--------------------------------------Part 1-------------------------------------
training_data = [
({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'no'}, False),
({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'yes'}, False),
({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'no'}, True),
({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'no'}, True),
({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'no'}, True),
({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'yes'}, False),
({'level':'Mid', 'lang':'R', 'tweets':'yes', 'phd':'yes'}, True),
({'level':'Senior', 'lang':'Python', 'tweets':'no', 'phd':'no'}, False),
({'level':'Senior', 'lang':'R', 'tweets':'yes', 'phd':'no'}, True),
({'level':'Junior', 'lang':'Python', 'tweets':'yes', 'phd':'no'}, True),
({'level':'Senior', 'lang':'Python', 'tweets':'yes', 'phd':'yes'}, True),
({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'yes'}, True),
({'level':'Mid', 'lang':'Java', 'tweets':'yes', 'phd':'no'}, True),
({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'yes'}, False)
]

test_ = ({"level": "Junior","lang": "Java","tweets": "yes","phd": "no"},
{"level": "Junior","lang": "Java","tweets": "yes","phd": "yes"}, {"level": "Intern"},
{"level": "Senior"}
 )

def part_1():

    target = []
    data_ = []
    for item in training_data:
        target.append(item[1])
        data_.append(item[0])
    data = pd.DataFrame(target)
    y = data[data.columns[-1]]
    x = pd.DataFrame(data_)
    tree = DecisionTreeID3(max_depth = 3)
    tree.fit(x,y)
    x_test = pd.DataFrame(test_)
    n = len(test_)
    result = tree.classify(x_test,n)
    for i in range(0,n):
```

```
        if result[i] == True:
            Str = "Hire"
        else:
            Str = "Do not Hire"
        print(test_[i],"   ", Str)
    print("\n")
#---------------------------------Part 2---------------------------------

    filename_1 = "train.csv"

    def raw_data():
      data = pd.read_csv(filename_1)

      train = data.iloc[:650, ]
      test = data.iloc[650:,  :]
      y = train[train.columns[1]]
      drop_elements = ['PassengerId', 'Name', 'Ticket', 'Cabin', 'SibSp','Survived']
      x = train.drop(drop_elements, axis = 1)
      tree = DecisionTreeID3(max_depth = 5, min_samples_split = 2, min_gain = 1e-4)
      tree.fit(x,y)

      y_test = test[test.columns[1]].to_list()
      x_test = test.drop(drop_elements, axis = 1)
      n = len(y_test)
      result = tree.classify(x_test,n)
      acc = 0

      for i in range(n):
          if int(result[i]) == int(y_test[i]):
                  acc += 1
      print(" The total accuracy in raw test set: ",acc/n)

    #Process data to get better performence
    def process_data(filename):

       data = pd.read_csv(filename)

       #create a new column named: Has_cabin based on Cabin
       data['Has_Cabin'] = data["Cabin"].apply(lambda x: 0 if type(x) == float else 1)

       # Create new feature FamilySize as a combination of SibSp and Parch
       data['FamilySize'] = data['SibSp'] + data['Parch'] + 1

       # Create new feature IsAlone from FamilySize
       data['IsAlone'] = 0
       data.loc[data['FamilySize'] == 1, 'IsAlone'] = 1

       # Mapping Sex
       data['Sex'] = data['Sex'].map( {'female': 0, 'male': 1} ).astype(int)

       # Mapping Fare
       data['Fare'] = data['Fare'].fillna(data['Fare'].median())
       data.loc[ data['Fare'] <= 7.91, 'Fare']                                    = 0
       data.loc[(data['Fare'] > 7.91) & (data['Fare'] <= 14.454), 'Fare'] = 1
       data.loc[(data['Fare'] > 14.454) & (data['Fare'] <= 31), 'Fare']   = 2
       data.loc[ data['Fare'] > 31, 'Fare']
= 3
       data['Fare'] = data['Fare'].astype(int)

       #Mapping Embark
       data['Embarked'] = data['Embarked'].fillna('S')
```

```
        data['Embarked'] = data['Embarked'].map( {'S': 0, 'C': 1, 'Q': 2} ).astype(int)

        # Mapping Age
        age_avg = data['Age'].mean()
        data.loc[np.isnan(data['Age']), 'Age'] = age_avg
        data['Age'] = data['Age'].astype(int)
        data.loc[ data['Age'] <= 16, 'Age']                                          = 0
        data.loc[(data['Age'] > 16) & (data['Age'] <= 32), 'Age'] = 1
        data.loc[(data['Age'] > 32) & (data['Age'] <= 48), 'Age'] = 2
        data.loc[(data['Age'] > 48) & (data['Age'] <= 64), 'Age'] = 3
        data.loc[ data['Age'] > 64, 'Age'] = 4

        drop_elements = ['PassengerId', 'Name', 'Ticket', 'Cabin', 'SibSp']
        data = data.drop(drop_elements, axis = 1)

        return data

    def new_data():

      data = process_data(filename_1)

      train = data.iloc[:650, ]
      test = data.iloc[650:,  :]

      y = train[train.columns[0]]
      x = train.drop(["Survived"], axis = 1)
      tree = DecisionTreeID3(max_depth = 5, min_samples_split = 2, min_gain = 1e-4)
      tree.fit(x,y)

      y_test = test[test.columns[0]].to_list()
      x_test = test.drop(["Survived"], axis = 1)
      n = len(y_test)
      result = tree.classify(x_test,n)
      acc = 0

      for i in range(n):
         if int(result[i]) == int(y_test[i]):
                  acc += 1
      print(" The total accuracy in cleaned test set: ",acc/n)
#---------------------------------Program Main---------------------------------
def main():
    print("The result of part1: \n")
    part_1()
    print("The result of part2: \n")
    raw_data()
    new_data()
main()
#--------------------------------End of Program--------------------------------
```