Data Science

Assignment2 : Decision Tree

2016025532 컴퓨터전공 심수정

1. Summary of Algorithm

decision tree 는 어떠한 데이터를 입력받았을 때 tree 의 형태에 맞추어 어떤 class 에 속할지 알아내는 algorithm 입니다. 이를 위해 tree 를 만들기 위한 training set 이 필요합니다.

Decision tree 를 만들기 위하여 training set 과 gain ration 를 활용하게 됩니다. Training set 의 attribute 들을 받고, 이중 가장 잘 나누는 것을 기준점으로 삼아 tree 의 label 을 설정해 나누게 되는데 가장 잘 나누는 것을 찾기 위해 gain ratio 를 활용합니다.

Gain ratio 는 다음과 같이 계산합니다.

$$GainRatio(A) = Gain(A)/SplitInfo(A)$$

여기서 information gain 은

$$\text{Gain} = Info(D) - Info_a(D)$$

Info 는

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2 p_i$$

$Info_a$는

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} * Info(D_j)$$

으로 계산됩니다.

Split info 는

$$SplittedInfo = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} * \log_2 \frac{|D_j|}{|D|}$$

으로 계산할 수 있습니다.

이렇게 계산된 gain ration 중 최대값을 가지는 attribute 가 선택되고, 이를 tree 의 현재 node 에서 자식 node 로 나누는 기준으로 활용하게 됩니다. 더 이상 나누어 지지 않을 때까지 이 과정을 반복하게 되고, 이를 활용하여 어떤 class 에 test data 가 속할지 예측하게 됩니다.

2. Detailed Description of Code

```
get_class()  while current_c…
```

```python
import sys
from math import log

# get name of files from argument
training_file = sys.argv[1]
test_file = sys.argv[2]
result_file = sys.argv[3]
```

실행시 넘겨받은 argument 로부터 training file name, test file name, output file name 을 변수로 받아옵니다.

```python
# parsing file to attribute list, data list
# attribute_list : {'attribute name' : 'attribute value'}
# data : [data]
def parse_file_to_attribute_list (filename) :
    # read file and parsing it to data
    with open(filename) as file :
        attributes = file.readline().split()
        data = file.read().split()
        data = [data[i : i + len(attributes)] for i in range(0, len(data), len(attributes))]

    # generate attribute dictionary
    attribute_list = {}
    for i in attributes :
        attribute_list[i] = []

    # get attribute list
    for i in data :
        for j in range(len(attributes)) :
            if i[j] not in attribute_list[attributes[j]] :
                attribute_list[attributes[j]].append(i[j])

    return attribute_list, data
```

파일의 이름을 넘겨받아 해당 파일을 열고, 어떠한 attribute 들이 있는지, 또한 어떠한 data 들이 있는지 각각 리스트의 형태로 받아옵니다.

그리고 attribute list 라는 dictionary 를 만들어 attribute 의 이름별로 data 에 어떠한 속성들이 있는지 받아옵니다. 마지막으로 이 attribute list 라는 dictionary, data 를 return 해줍니다.

```python
# generate decision tree
def generate_decision_tree (attribute_list, data_set) :
    tree = [[[{}, [], data_set]]] # [tree [level [node]]]
                                  # node : {label}, [child label of class], [data list]

    # make data homogeneous by splitting
    level = 0
    while True :
        count = 0
        new_level_nodes = [] # [level [node]]
        for i in tree[level] :
            label = get_label(i[0], i[2], attribute_list, tree)

            if label == None : # no more label or homogenous
                count = count + 1
                continue

            i[1].append(label)
            new_level_nodes.extend(split_data(i[0], label, attribute_list[label], list(attribute_list.keys()).index(label), i[2]))
        tree.append(list(new_level_nodes))
        if count == len(tree[level]) :
            break
        level = level + 1


    # if tree node has no data -> remove
    # if tree node homogenous -> child label location = class
    reduced_tree = tree.copy()
    tree.reverse()

    index = len(tree) - 1
    for i in tree :
        position = 0
        for j in i :
            if len(j[1]) <= 0 : # if child node
                majority_class = get_majority(j[2], list(attribute_list.values())[-1])
                if majority_class == None : # no data
                    #   parent_idx, parent_pos = get_parent_index(j[0], reduced_tree)
                    #   reduced_tree[parent_idx][parent_pos][1] = get_majority(reduced_tree[parent_idx][parent_pos][2], list(attribute_list.values())[-1])
                    reduced_tree[index].remove(j)
                    continue
                reduced_tree[index][position][1].append(majority_class)
            position = position + 1
        index = index - 1
    return reduced_tree
```

tree 를 [tree [level [node [{labels}, [child label or class], [data]]]]]의 형태로 생성하고, level 별로 for 문을 돌며 새롭게 나눌 label 을 받습니다. 이 label 을 labels 위치에 추가하고, 원래 label(parent label)도 이 dictionary 에 update

해줍니다. 만약 label 이 None 이라면 바로 다른 node 로 넘어가 이를 반복합니다. 이때, level 의 모든 node 가 다 None 을 return 한다면, 즉, 이제 더 이상 나눠지지 않는다면 이 작업을 끝냅니다.

이제 반대로 tree 를 돌며 data 가 없는 node 가 존재한다면 이를 없애줍니다. 그리고 이 작업을 마친 tree 를 return 해줍니다.

```
def get_label (used_label, data_set, attribute_list, tree) :
    label_candidate = list(set(list(attribute_list.keys())[:-1])-set(used_label))

    # list for saving gini values
    gain_ratio = {}
    compare = []
    for _ in label_candidate :
        compare.append(0)
    gain = gain_ratio.copy()

    # no more usable label, return None
    if len(compare) <= 0 :
        return None

    # calculate gini each labels
    info_val = get_info(list(attribute_list.keys())[-1], list(attribute_list.values())[-1], data_set)

    for i in label_candidate :
        gain[i] = get_gain(info_val, i, attribute_list[i], list(attribute_list.keys()).index(i), list(attribute_list.keys())[-1], list(attribute_list.values())[-1], data_set)
        splitted = get_splitted_info(i, attribute_list[i], list(attribute_list.keys()).index(i), data_set)
        if splitted != 0 :
            gain_ratio[i] = gain[i] / splitted
        else :
            gain_ratio[i] = gain[i]

    # find homogenous or return max label by gain ratio
    if max(gain_ratio.values()) == 1 :
        return None
    elif list(gain_ratio.values()) == compare :
        return None
    else :
        new_label = label_candidate[list(gain_ratio.values()).index(max(list(gain_ratio.values())))]
        return new_label
```

새롭게 나눠줄 label 을 return 합니다. 이 때 쓰여진 label 들과 전체 label 을 통하여 안쓰여진 label 을 먼저 label_candidate 에 저장하고, 이 개수만큼 for 문을 돌면서 0 만 가지는, 모든 gain 이 0 이 나오는 경우를 확인하기 위한 리스트를 만들어줍니다. 이제 label candidate 들의 gain, splitted info 를 얻어 gain ratio 를 얻고, 최대값이 1 인 경우는 homogenous 한 경우이므로 None 을, 모두다 0 이 나온 경우는 data 가 없는 경우이므로 None 을, 그 외의 경우는 max 값을 가지는 label 을 return 해줍니다.

```
def get_info (class_label, class_attribute, data_set) :
    # count by class attribute
    count = [0 for _ in class_attribute]
    for i in data_set :
        count[class_attribute.index(i[-1])] += 1

    # if no data, return None
    total = len(data_set)
    if total <= 0 :
        return 0

    # calculate info
    info = 0
    for i in count :
        if i != 0 :
            info -= ((i/total)*log(i/total, 2))
    return info
```

class attribute 들과 data set 을 받아 info 를 위의 info 식에 맞춰 계산합니다. 그리고 얻어진 info 값을 return 해줍니다.

```
def get_attribute_info (label, label_attribute, label_index, class_label, class_attribute, data_set) :
    count_by_label_attribute = [0 for _ in label_attribute]
    data_set_by_label_attribute = [[] for _ in label_attribute]

    # count data by label attribute
    for i in data_set :
        count_by_label_attribute[label_attribute.index(i[label_index])] += 1
        data_set_by_label_attribute[label_attribute.index(i[label_index])].append(i)

    if sum(count_by_label_attribute) <= 0 : # no data
        return 0

    info_a = 0
    index = 0
    for i in count_by_label_attribute :
        info_a += (i/sum(count_by_label_attribute) * get_info(class_label, class_attribute, data_set_by_label_attribute[index]))
        index = index + 1

    return info_a
```

특정 label 에 attribute 에 해당하는 info gain 을 계산하였을 때 어떤 info gain 이 나오는지 위의 식에 맞추어 계산합니다. 그리고 계산된 값을 return

해줍니다.

```
def get_gain (info, label, label_attribute, label_index, class_label, class_attribute, data_set) :
    return info - get_average_info(label, label_attribute, label_index, class_label, class_attribute, data_set)
```

info 에서 특정 label 의 attribute 에 대해 info 를 계산한 값을 빼서 return 해줍니다.

```
def get_splitted_info (label, label_attribute, label_index, data_set) :
    count_by_label_attribute = [0 for _ in label_attribute]

    # count data by label attribute
    for i in data_set :
        count_by_label_attribute[label_attribute.index(i[label_index])] += 1

    # if no data, return None
    total = len(data_set)
    if total <= 0 :
        return 0

    # calculate info
    info = 0
    for i in count_by_label_attribute :
        if i != 0 :
            info -= ((i/total)*log(i/total, 2))
    return info
```

얼마만큼의 data 가 해당 label 의 attribute 들에 대해 잘 나누어지는지를 위의 식대로 계산하여 return 해줍니다.

```
def split_data (parent_label, new_label, new_label_attributes, new_label_index, data_set) :
    new_level_nodes = [[{}, [], []] for _ in new_label_attributes]

    # append label
    index = 0
    for i in new_level_nodes :
        i[0][new_label] = new_label_attributes[index]
        i[0].update(parent_label)
        index = index + 1

    # append data
    for i in data_set :
        new_level_nodes[new_label_attributes.index(i[new_label_index])][2].append(i)

    return new_level_nodes
```

받은 label 에 맞추어 데이터를 나누고, 이를 같은 tree level 의 node 를 저장하는 list 에서, 같은 attribute 를 가지는 node 의 data 모임에 추가해줍니다. 그리고 이 새로운 level 을 return 해줍니다.

```
# if leaf node get class by majority
def get_majority (data_set, class_attribute) :
    count = [0 for _ in class_attribute]
    origin_count = count.copy()

    for i in data_set :
        count[class_attribute.index(i[-1])] += 1

    if count == origin_count : # no data
        return None

    max_count = 0
    for i in count :
        if i == max(count) :
            max_count = max_count + 1

    return class_attribute[count.index(max(count))]
```

특정 node 의 data set 중에서 가장 많이 나오는 class 가 어떤 class 인지를 찾습니다. 만약 확인하는 도중 data 가 하나도 없다면 None 을 return 하고, 그렇지 않다면 count 횟수가 가장 높은 값을 return 하되, 만약 동등한 값이 있다면 더 먼저 저장되어있는 class 를 return 합니다.

```
def get_parent (label, tree) :
    label_copy = label.copy()
    del label_copy[list(label_copy.keys())[0]]

    for i in tree :
        for j in i :
            if j[0] == label:
                return i

    return get_parent(label_copy, tree)
```

현재 node 의 label 을 통하여 parent node 를 찾아 return 합니다. 만약 현재 노드의 부모 노드가 찾아지지 않는다면 라벨에서 가장 하위 level 에 속하는 attribute 를 지우고, 다시 parent node 를 찾습니다.

```
def get_parent_index (label, tree) :
    label_copy = label.copy()
    del label_copy[list(label_copy.keys())[0]]

    index = 0
    position = 0
    for i in tree :
        position = 0
        for j in i :
            if j[0] == label:
                return index, position
            position = position + 1
        index = index + 1

    return get_parent_index(tree, label_copy)
```

위의 get_parent 함수와 같은 방식으로 작동하되, 이는 parent node 가 아닌 tree 에서의 parent node 의 index 를 return 합니다.

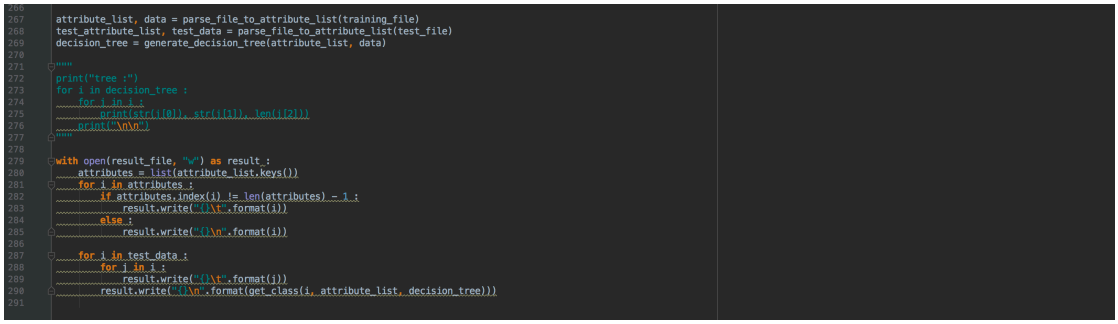
```
def get_class (data, attribute_list, tree) :
    # initial search settings
    label = {}

    # tree search
    for i in tree : # each level in tree
        for j in i : # each node in level
            if label.items() <= j[0].items() : # if all label in current node label
                if len(j[1]) <= 0 : # no next label
                    parent_label = label.copy()
                    current_class = None
                    while current_class == None :
                        parent = get_parent(parent_label, tree)
                        current_class = get_majority(parent[2], list(attribute_list.values())[-1])
                        parent_label = parent[0]
                    return current_class
                    #return get_majority(j[2], list(attribute_list.values())[-1])
                elif j[1][0] in list(attribute_list.values())[-1] : # if child label == class (leaf node)
                    return j[1][0]
                else : # get next level
                    label = {}
                    label[j[1][0]] = data[list(attribute_list.keys()).index(j[1][0])]
                    label.update(j[0])
                    break

    parent_label = label.copy()
    current_class = None
    while current_class == None :
        parent = get_parent(parent_label, tree)
        current_class = get_majority(parent[2], list(attribute_list.values())[-1])
        parent_label = parent[0]
    return current_class
```

test file 이 어떤 class 에 속하는지 확인합니다. 만약 자식을 탐색하면서 다음 level 에 해당하는 값이 없는 경우는 parent node 를 탐색하여 parent node 에 어떤 class 에 속하는 경우가 많았는지 확인하고, 해당 node 가 class 정보를 들고 있다면 이를 return 합니다. 그렇지 않다면 다음 level 을 탐색합니다.

만약 이 과정을 통해 발견되지 않는다면 parent node 를 탐색하며 get majority 함수를 통하여 class 를 return 합니다.



```
attribute_list, data = parse_file_to_attribute_list(training_file)
test_attribute_list, test_data = parse_file_to_attribute_list(test_file)
decision_tree = generate_decision_tree(attribute_list, data)

"""
print("tree :")
for i in decision_tree :
    for j in i :
        print(str(j[0]), str(j[1]), len(j[2]))
    print('\n\n')
"""

with open(result_file, "w") as result :
    attributes = list(attribute_list.keys())
    for i in attributes :
        if attributes.index(i) != len(attributes) - 1 :
            result.write("{}\t".format(i))
        else :
            result.write("{}\n".format(i))

    for i in test_data :
        for j in i :
            result.write("{}\t".format(j))
        result.write("{}\n".format(get_class(i, attribute_list, decision_tree)))
```

처음에 training file 을 parsing 하여 attribute list, data 를 찾고, test file 도 parsing 하여 test file 의 data 를 저장합니다. 후에 tree 를 generate 하고, output file 을 열어 attribute 를 쓴 다음 test file 의 data 들을 output file 에 다시 쓰고, 이의 class 를 예측하여 함께 적어줍니다.

3. Instruction for Compiling Source Code

이 코드는 python3 을 기반으로 작성되었습니다. 따라서 python3 가 설치

되어 있어야 합니다.

"**python3 dt.py [training_file_name] [test_file_name] [output_file]**"의 형태로 complie, 실행하면 됩니다.

4. Other Sepcification of Implementation and Testing

training set 을 통해 만든 tree 의 각 child node 가 어떤 class 를 가지는지 알아내기 위해 어떤 class 가 가장 많이 나타나는 가를 사용하게 됩니다. 만약 homogenous 한 경우라면 당연히 하나의 class 만 존재하기 때문에 이가 다수가 되고, 그렇지 않은 경우라면 어떠한 class 가 많이 속했는지 알아내서, 이를 이 node 의 대표 class 로 분류하게 됩니다.

Tree 의 자료구조가 다소 복잡합니다. 우선 전체 tree 를 list 의 형태로 표현하는데 각 level 별로 서로 다른 list 에 속합니다. 여기까지 [[level0], [level1], [level2], ...] 의 형태입니다. 그리고 level 안의 각각의 node 도 list 로 표현됩니다. [[node1], [node2], [node3], ...]의 형태가 level n 에 해당되는 것입니다. 그리고 하나의 node 는 [{labels}, [child label or class], [data]]의 형태를 가지고 있습니다.

또한, parent node 를 구분하기 위하여 해당 node 의 dictionary 에 현재 자신의 분류 attribute 는 물론 부모들이 어떠한 attribute 로 분류되어 왔느냐까지 저장하게 됩니다. 예를 들어, 처음에 'a'라는 attribute 로 분류되었고, 그 밑에서 'b'라는 attribute 로 분류되었다면, 이 node 의 label dictionary 는 {'b' : 'b_value', 'a' : 'a_value'}가 됩니다.