

Programming Assignment #2

2017029916

양동해

1. 알고리즘 요약

Training dataset을 통해 Decision Tree를 만든 뒤, Test dataset의 class label을 찾아내는 프로그램이다. 코드는 총 3가지 파트로 구분되며 각 파트당 알고리즘은 다음과 같다.

- Node 클래스
- DecisionTree 클래스 (및 주요 함수)
 - 생성자
 - information_gain()
 - gain_ratio()
 - gini_index()
 - model_construct() (create_node() 함수에 주요 기능 포함)
 - mode_usage()
- 파일 읽기 및 Decision Tree 실행

2. 주요 코드 상세

코드를 위에서 언급한 3가지 파트로 나누어, 각 파트가 무엇을 하는지 기술하였다.

[Node 클래스]

```
# Node 클래스
class Node:
    def __init__(self, attribute=None, class_label_value=None):
        # attribute: internal node -> 자식으로 내려갈때 비교할 attribute를 나타냄
        # class_label_value: leaf node -> leaf node인 경우, 현재 아이템이 무슨 class인지 나타냄
        self.attribute = attribute
        self.class_label_value = class_label_value

        # child_node: internal node인 경우, 자식노드들을 저장
        self.child_node = {}

    def is_leaf(self): # leaf 노드인지 확인
        if len(self.child_node) == 0: return True
        else: return False

    def get_class_label_value(self, sample): # leaf 노드로 가서 class_label_value를 반환
        # leaf node인 경우
        if self.is_leaf(): return self.class_label_value

        # leaf node가 아닌 경우
        attribute_value = sample[self.attribute]
        for child, node in self.child_node.items(): # child node 순회하면서 이동할 node 찾기 (child: 튜플형태로 attribute value가 들어가 있음)
            if attribute_value in child:
                return node.get_class_label_value(sample)
```

Node 클래스는 DecisionTree 클래스에서 사용될 root 및 자식 노드들의 구조를 갖는 클래스이다. 변수는 attribute, class_label_value, child_node를 가진다. attribute는 해당 노드가 internal node일 때 attribute를 의미하고, class_label_value는 해당 노드가 leaf node일 때 현재 무슨 class label을 갖는지 나타내며, child_node는 자식 노드들을 담는 dictionary 변수이다. 이 외에 현재 노드가 leaf node인지 판단하는 is_leaf() 함수와, 자식 노드를 recursive하게 탐색해 leaf node에 도달할 경우 그 노드의 class label이 무엇인지 판단하는 get_class_label_value() 함수가 존재한다.

[DecisionTree 클래스]

- 생성자

```
# Decision Tree 클래스
class DecisionTree:
    def __init__(self, train_dataset, attribute_selection_measure):
        # attribute_selection_measure: attribute selection measure가 무엇인지 저장
        # 종류: information_gain, gain_ratio, gini_index
        self.attribute_selection_measure = attribute_selection_measure

        # attribute_values: attribute가 가진 value들의 종류들, column별로 array 형태로 저장
        # class_label: class label에 해당하는 attribute name 저장
        self.attribute_values = {}
        for attribute in train_dataset.columns[1:-1]:
            self.attribute_values[attribute] = train_dataset[attribute].unique()
        self.class_label = train_dataset.columns[-1]

        self.root_node = None # root node 초기화
        self.model_construct(train_dataset) # model construction: decision tree 만들기
```

DecisionTree 클래스는 classification을 하기 위한 모델을 만드는 클래스이다. 변수는 attribute_selection_measure, attribute_values, class_label, root_node를 가진다.

attribute_selection_measure는 attribute를 선택하기 위한 measure로 어떤 방법을 사용할지 저장하는 변수이고, attribute_values는 training dataset이 가지고 있는 모든 attribute들의 value들을 key(attribute 이름) : value(해당 attribute의 value들) 형태로 가지고 있는 변수이다. class_label은 training dataset의 class label 컬럼만 따로 가지고 있는 변수이며, root_node는 tree의 root를 가리키는 변수이다. 변수들의 초기화가 모두 끝나면 model_construct() 함수를 실행한다.

- information_gain()

```
''' information gain '''
def info(self, dataset):
    info_D = 0.0
    for class_label_value, C_i in dataset[self.class_label].value_counts().iteritems():
        p_i = C_i / len(dataset)
        info_D += (-1.0) * p_i * math.log2(p_i + 1e-9) # log2(0) 오류 막기 위해, trivial 값 add
    return info_D

def information_gain(self, attribute, dataset):
    gain_A, info_A_D = 0.0, 0.0
    for attribute_value in self.attribute_values[attribute]:
        new_dataset = dataset[dataset[attribute] == attribute_value]
        info_A_D += (len(new_dataset) / len(dataset)) * self.info(new_dataset)
    gain_A = self.info(dataset) - info_A_D
    return gain_A
```

information_gain() 함수는 attribute selection measure로 사용될 information gain을 구현한 함수이며, 공식을 그대로 코드로 구현했기 때문에 자세한 설명은 생략한다.

- gain_ratio()

```
''' gain ratio '''
def split_info(self, attribute, dataset):
    splitinfo_A_D = 0.0
    for attribute_value in self.attribute_values[attribute]:
        new_dataset = dataset[dataset[attribute] == attribute_value]
        p_i = len(new_dataset) / len(dataset)
        splitinfo_A_D += (-1.0) * p_i * math.log2(p_i + 1e-9) # log2(0) 오류 막기 위해, trivial 값 add
    return splitinfo_A_D

def gain_ratio(self, attribute, dataset):
    gain_A = self.information_gain(attribute, dataset)
    splitinfo_A_D = self.split_info(attribute, dataset)
    return gain_A / splitinfo_A_D
```

gain_ratio() 함수는 attribute selection measure로 사용될 gain ratio를 구현한 함수이며, 공식을 그대로 코드로 구현했기 때문에 자세한 설명은 생략한다.

- gini_index()

```
''' gini index '''
def gini(self, dataset):
    gini_D = 1.0
    for class_label_value, C_i in dataset[self.class_label].value_counts().iteritems():
        p_i = C_i / len(dataset)
        gini_D -= p_i ** 2
    return gini_D

def gini_index(self, attribute, dataset):
    # value_comb: value들을 binary partition으로 나눈 리스트
    attribute_value = self.attribute_values[attribute]
    value_comb = []
```

```

for i in range(1, len(attribute_value)):
    if i*2 > len(attribute_value): break
    lefts = list(map(set, list(combinations(attribute_value, i))))
    for left in lefts:
        right = set(attribute_value) - left
        value_comb.append(tuple([tuple(left), tuple(right)]))
# print(value_comb)

# binary partition 선택
asm_dict = {}
gini_D = self.gini(dataset)
for comb in value_comb:
    d1 = dataset[dataset[attribute].isin(comb[0])]
    d2 = dataset[dataset[attribute].isin(comb[1])]

    d1_D = len(d1) / len(dataset)
    d2_D = len(d2) / len(dataset)

    d1_gini = d1_D * self.gini(d1)
    d2_gini = d2_D * self.gini(d2)

    asm_dict[comb] = gini_D - (d1_gini + d2_gini) # gini index 적용
asm_dict = sorted(asm_dict.items(), key=operator.itemgetter(1), reverse=True) # measure 값이 큰 것부터 내림차순
# print(asm_dict[0])

return asm_dict[0]

```

gini_index() 함수는 attribute selection measure로 사용될 gini index을 구현한 함수이다. gini index는 자식 노드를 두개만 갖기 때문에, 해당 attribute의 value들을 binary partiton으로 나누어야 한다. binary partition으로 나눈 후보들은 value_comb 리스트에 들어가 있고, 이들 모두의 gini index를 구한 뒤, 가장 높은 gini_index값을 갖는 binary partition을 return 한다.

- model_construct()

```

def model_construct(self, train_dataset): # train dataset을 받았으면, 그 샘플들에 대해 모델을 생성
    self.root_node = self.create_node(train_dataset)

```

model_construct() 함수는 node를 create 하는 것부터 시작하므로, create_node() 함수가 모델을 생성하는 함수로 볼 수 있다.

- create_node()

```

def create_node(self, dataset):
    # leaf node 처리
    if dataset[self.class_label].nunique() == 1: return Node(None, dataset[self.class_label].unique()[0]) # 남아있는 sample들이 전부 같은 클래스인 경우 (unique를 통해 확인)
    if len(dataset.columns) == 1: return Node(None, self.get_majority(dataset)) # 남아있는 attribute가 없는 경우 (class label column만 존재하는 경우) but, del 하지 않으므로 실행 안해도 됨

    # attribute 선택해서 노드 생성
    node = None
    selected_attribute = self.select_attribute(dataset)
    if self.attribute_selection_measure == "information_gain" or self.attribute_selection_measure == "gain_ratio": node = Node(selected_attribute, None)
    elif self.attribute_selection_measure == "gini_index": node = Node(selected_attribute[0], None)
    # print(selected_attribute)

    # child 생성
    if self.attribute_selection_measure == "information_gain" or self.attribute_selection_measure == "gain_ratio":
        ''' ver1-1 (information gain, gain ratio) '''
        # for attribute_value in self.attribute_values[selected_attribute]:
        #     new_dataset = dataset[dataset[selected_attribute] == attribute_value] # 선택된 attribute value들로 dataset 필터링
        #     del new_dataset[selected_attribute]
        #     # print(new_dataset)
        #     if len(new_dataset) == 0: return Node(None, self.get_majority(dataset)) # new_dataset에 남아있는 sample이 없는 경우
        #     else: node.child_node[attribute_value] = self.create_node(new_dataset)
        ''' ver1-2 (information gain, gain ratio + gini_index) '''
        child_values = self.gini_index(selected_attribute, dataset)
        # print(child_values)
        for child_value in child_values[0]:
            new_dataset = dataset[dataset[selected_attribute].isin(list(child_value))] # 선택된 attribute value들로 dataset 필터링
            # del new_dataset[selected_attribute] # 실행 안하는 것이 더 성능 좋음 => 아래 단계에 갔을 때도, 같은 attribute로 나눌 수 있어야 좀 더 분류 잘 함
            # print(new_dataset)
            if len(new_dataset) == 0: return Node(None, self.get_majority(dataset)) # new_dataset에 남아있는 sample이 없는 경우
            else: node.child_node[child_value] = self.create_node(new_dataset)
    elif self.attribute_selection_measure == "gini_index":
        ''' ver2 (gini index) '''
        # print(selected_attribute[1])
        for child_value in selected_attribute[1][0]:
            new_dataset = dataset[dataset[selected_attribute[0]].isin(list(child_value))] # 선택된 attribute value들로 dataset 필터링
            # del new_dataset[selected_attribute[0]] # 실행 안하는 것이 더 성능 좋음 => 아래 단계에 갔을 때도, 같은 attribute로 나눌 수 있어야 좀 더 분류 잘 함
            # print(new_dataset)
            if len(new_dataset) == 0: return Node(None, self.get_majority(dataset)) # new_dataset에 남아있는 sample이 없는 경우
            else: node.child_node[child_value] = self.create_node(new_dataset)

    return node

```

create_node() 함수는 처음에 dataset의 크기에 따라 leaf node 처리를 해주는 코드가 있으며, leaf node가 아닐 시에 attribute를 선택하여 node를 생성한다. attribute를 선택하는 과정은 위에서 구현한 3가지 measure 중 하나로 실행되며, 실행한 measure에 맞게 child를 생성하고, 생성하는 방법은 create_node() 함수를 recursive하게 호출하여 생성한다.

child 생성 파트는 크게 ver1-1, ver1-2, ver2로 구분되어 있는데, ver1-1은 information gain 또는 gain ratio를 이용해 attribute를 select 했을 경우이고, ver2는 gini index를 이용해 attribute를 select 했을 경우이다. ver1-2는 attribute를 선택하는 과정에선 information gain 또는 gain ratio를 이용했지만, child를 선택하는 과정에선 gini index를 이용해 자식 노드를 2개로만 만드는 경우이다. 이렇게 구현을 한 이유는 최적의 성능을 찾기 위해 여러가지 시도를 한 결과이고, 이는 "3. 컴파일 및 실행" 파트에서 볼 수 있다.

그리고 gini index로 child를 생성할 때, 조건에 맞게 분리된 새로운 dataset을 넘겨주는데, 이때 test attribute로 선택된 attribute를 dataset에서 삭제하지 않는다. 이렇게 구현하게 되면 이전에 선택된 attribute가 자식 노드에서 다시 한번 선택될 가능성이 생기지만, 실제 delete를 할 때와 안 할 때의 성능차이를 확인한 결과, 안 할 때 성능이 더 좋게 나와서, 삭제를 하는 라인을 주석 처리 했다. 이 성능차이 또한 "3. 컴파일 및 실행" 파트에서 볼 수 있다.

- mode_usage()

```
def mode_usage(self, test_dataset): # test dataset을 받았으면, 그 샘플들에 대해 모델을 통해 나온 class label을 생성
    class_label_list = []
    for i in range(len(test_dataset)):
        sample = test_dataset.loc[i]
        class_label_list.append(self.root_node.get_class_label_value(sample))
    test_dataset[self.class_label] = class_label_list
    return test_dataset
```

mode_usage() 함수는 test dataset을 받아서, 각 행들에 대해 class label을 찾아주는 함수이다. 찾는 방법은 root에서 leaf로 내려가며 알아낸다.

[파일 읽기 및 Decision Tree 실행]

```
# 파일 읽기
train_filename = sys.argv[1] # dt_train.txt
test_filename = sys.argv[2] # dt_test.txt
output_filename = sys.argv[3] # dt_result.txt
train_dataset = pd.read_csv(train_filename, sep="\t")
test_dataset = pd.read_csv(test_filename, sep="\t")

# Decision Tree 실행
# 종류: information_gain, gain_ratio, gini_index
dt = DecisionTree(train_dataset, "information_gain")
result = dt.mode_usage(test_dataset)
result.to_csv(output_filename, index=False, sep="\t")
```

이 파트는 인자로 입력 받은 파일들을 읽고, Decision Tree를 만들어 모델을 사용하는 단계이다.

3. 컴파일 및 실행

- Python version: 3.9.12 + (pandas library 설치 필수; 설치 커맨드: pip3 install pandas)

- 프로그램 실행: python3 dt.py [train data file name] [test data file name] [result file name]

```
eastsea@EastSeai-iMac assignment2 % python3 --version
Python 3.9.12
eastsea@EastSeai-iMac assignment2 % ls
dt.py          dt_answer.txt  dt_answer1.txt dt_test.exe    dt_test.txt    dt_test1.txt    dt_train.txt    dt_train1.txt
eastsea@EastSeai-iMac assignment2 % python3 dt.py dt_train.txt dt_test.txt dt_result.txt
eastsea@EastSeai-iMac assignment2 % mono dt_test.exe dt_answer.txt dt_result.txt
5 / 5
eastsea@EastSeai-iMac assignment2 %
```

- "dt_train1.txt"와 "dt_test1.txt"로 실행할 경우 (ver1-2: information gain + gini index 이용)

```
eastsea@EastSeai-iMac assignment2 % python3 dt.py dt_train1.txt dt_test1.txt dt_result1.txt
eastsea@EastSeai-iMac assignment2 % mono dt_test.exe dt_answer1.txt dt_result1.txt
338 / 346
eastsea@EastSeai-iMac assignment2 %
```

* 다른 measure를 이용할 경우 (measure와 ver을 바꿔가며 차례로 수행한 결과)

- ver1-1: information gain 이용 => 315 / 346

- ver1-1: gain ratio 이용 => 318 / 346

- ver2: gini index (attribute delete 진행 O) 이용 => 294 / 346
- ver2: gini index (attribute delete 진행 X) 이용 => 336 / 346
- ver1-2: information gain + gini index 이용 => 338 / 346
- ver1-2: gain ratio + gini index 이용 => 335 / 346

```
eastsea@EastSeai-iMac Programming_Assignment2 % python3 dt.py dt_train1.txt dt_test1.txt dt_result1.txt
eastsea@EastSeai-iMac Programming_Assignment2 % mono dt_test.exe dt_answer1.txt dt_result1.txt
315 / 346
eastsea@EastSeai-iMac Programming_Assignment2 % python3 dt.py dt_train1.txt dt_test1.txt dt_result1.txt
eastsea@EastSeai-iMac Programming_Assignment2 % mono dt_test.exe dt_answer1.txt dt_result1.txt
318 / 346
eastsea@EastSeai-iMac Programming_Assignment2 % python3 dt.py dt_train1.txt dt_test1.txt dt_result1.txt
eastsea@EastSeai-iMac Programming_Assignment2 % mono dt_test.exe dt_answer1.txt dt_result1.txt
294 / 346
eastsea@EastSeai-iMac Programming_Assignment2 % python3 dt.py dt_train1.txt dt_test1.txt dt_result1.txt
eastsea@EastSeai-iMac Programming_Assignment2 % mono dt_test.exe dt_answer1.txt dt_result1.txt
336 / 346
eastsea@EastSeai-iMac Programming_Assignment2 % python3 dt.py dt_train1.txt dt_test1.txt dt_result1.txt
eastsea@EastSeai-iMac Programming_Assignment2 % mono dt_test.exe dt_answer1.txt dt_result1.txt
338 / 346
eastsea@EastSeai-iMac Programming_Assignment2 % python3 dt.py dt_train1.txt dt_test1.txt dt_result1.txt
eastsea@EastSeai-iMac Programming_Assignment2 % mono dt_test.exe dt_answer1.txt dt_result1.txt
335 / 346
eastsea@EastSeai-iMac Programming_Assignment2 %
```