load(path)함수

training 파일에 attribute name들과 attribute values들로 이루어진 sample들을 형식에 맞게 바꿔서 필요한 값들을 반환하는 함수이다.

첫 줄에 attribute name들과 class name이 있는데 텍스트 파일 그대로의 첫 줄을 keys에 담아 나중에 결과를 출력할 때 쓰기 위해 반환한다. attribute_keys에는 attribute name들을 list로 담는다.

두번째 줄부터는 한 줄 씩 lines에 담아서 Sample들을 attribute_keys(attribute name list):attribute values(str_line에서 '₩t'와 끝에 class_value를 제외한 리스트)의 dict형식과 class value 값을 쌍으로 이루도록하여 Datas 리스트에 담아 반환한다.

이때 class values들을 set으로 중복을 제거하여 class_values 리스트에 담아서 반환한다.

dt.py > ...

```python
import math
from collections import Counter, defaultdict
from sys import argv

def load(path):
    Datas = []
    f = open(path, "r")
    attribute_keys = f.readline()
    keys = attribute_keys
    attribute_keys = attribute_keys.split('\t')
    n = len(attribute_keys)
    class_key = attribute_keys[n-1]
    class_values = []
    attribute_keys = attribute_keys[:n-1]
    for lines in f:
        str_line = list(lines.split('\t'))
        class_value = str_line.pop()
        class_value = class_value.strip('\n')
        Data = dict(zip(attribute_keys, str_line))
        Datas.append((Data, class_value))
        class_values.append(class_value)
    class_values = list(set(class_values))

    return Datas, class_values, keys

path = argv[1]

inputs, class_values, keys = load(path)


```

위의 세 반환 값, Datas, class_values리스트와 keys string을 각각 inputs, class_values, keys 전역변수에 담는다.

entropy

```
def entropy(class_probabilites):
# 확률이 0인 경우는 제외함
    return sum(-p * math.log(p ,2) for p in class_probabilites if p is not 0)
```

$$\sum_{i=1}^{class\ num} -p_i \log p_i$$

를 구하는 함수이다. 이는 Info(D)이다.

Class_probablities는 p리스트이다.

p_i는 현재 data들의 값을 볼 때 i번째 class에 포함될 확률이다.

class_probabilities

```
def class_probabilities(labels):
# 총 개수 계산
    total_count = len(labels)
    return [float(count) / float(total_count) for count in Counter(labels).values()]
```

p_i를 구하는 함수이다. (i번째 class value에 해당하는 수)/(전체 data 개수)이다.

labels = [no:2, yes:5]일 때

Counter(labels).values()는 [2, 5]를 의미한다.

$$p_i = \frac{|C_{i,D}|}{|D|}$$

data_entropy

```
def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
    return entropy(probabilities)
```

labels = [no:2, yes:5]이라 가정하자.

Probabilities = [2/7, 5/7]이다.

이러한 p_i들에 대한 Info값을 반환한다.

partition_entropy

```
def partition_entropy(subsets):
    total_count = sum(len(subset) for subset in subsets)
    return sum(data_entropy(subset) * len(subset) / total_count for subset in subsets)
```

Subsets들은 리스트들이다. 이 리스트 하나는 어떠한 attribute으로 split하였을 때 partition이다.

이러한 partition들의 총 entropy를 반환한다.

$$\sum_{i=1}^{attribute\ value\ num} \frac{|D_i|}{|D|} Info(D_i)$$

D_i는 i번째 partition 즉 subset에 해당한다. Subsets안에 subset의 개수는 attribute value의 개수와 같다.

partition_by

```
def partition_by(inputs, attribute):
    groups = defaultdict(list)
    for input in inputs:
        key = input[0][attribute]
        groups[key].append(input)
    return groups
```

Defaultdict를 써서

groups[attribute value(defaultdict의 key)]=[attribute value를 가지는 data들 리스트(defaultdict의 value)]의 형태로 inputs을 attribute이라는 attribute name으로 split 했을 때 partition들을 group에 담는다.

예) groups["yes"(student value)] = [{age: <30 , student: "yes", hungry: "yes"}, {age: 30, student: "yes", hungry: "no"}] [{}, {}] => group.values()

partition_entropy_by

```
def partition_entropy_by(inputs, attribute):
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())
```

Sample들을 담는 리스트 inputs과 attribute name에 해당하는 attribute를 인자로 받는다.
Inputs를 attribute를 기준으로 나누어 partitions에 담는다.
이러한 partition들의 총 entropy를 partition_entropy(partitions.values())로 반환한다.

build_tree

```
from functools import partial
def build_tree(inputs, split_candidates=None):
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()
```

inputs와 split_candidates를 인자로 가지고 매개변수 split_candidates에 해당하는 변수가 없이 inputs매개변수만 있다면 split_candidates는 default parameter 값으로 None값으로 설정한다.

그리고 이러한 기본값에 대응하여 split_candidates들을 inputs안에 한 sample에 대한 attribute_keys로 설정한다.

inputs형식이 [({attribute_keys: values}, class values), (), (), ...] 이므로 inputs[0][0].keys()는 attribute_keys가 된다.

```
    num_inputs = len(inputs)

    class_value = inputs[0][1]
    boolean = True
    for k in range(num_inputs):
        if class_value != inputs[k][1]:
            boolean = False
            break
    if boolean == True:
        return class_value
```

num_inputs는 현재 tree의 단계에서 트리 하위 단계의 data sample개수이다.
그 하위 모든 data sample들의 class value, 즉 inputs[k][1]들의 값이 다 같다면 현재 단계에서

class_value값을 반환하고 트리 생성을 멈춘다.

```
    if not split_candidates:
    # 다수결
        list = [inputs[k][1] for k in range(num_inputs)]
        leaf = max(list, key=list.count) #[("class value", 개수)]
        return leaf
```

만약 현재 단계에서 split_candidates 리스트가 빈 리스트라면 더이상 split할 수 있는 attribute이 없으므로 most voted로 class value를 결정하고 트리 생성을 멈춘다.

Data들의 모든 class value가 같은 경우, data에 더이상 split할 수 있는 attribute key가 없지만 class value는 섞여 있는 경우를 위에서 반환했다.
남은 경우는 가장 적합한 attribute로 split하는 것이다.

```
    best_attribute = min(split_candidates, key=partial(partition_entropy_by, inputs))
    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates if a != best_attribute]

    subtrees = { attribute_value : build_tree(subset, new_candidates) for attribute_value, subset in partitions.items()}
```

Split_candidates 리스트 안에 값들 중에 partition_entropy_by(inputs, split_candidates[k]) 값이 제일 작은 값을 best_attribute에 담는다. 이 값을 총 entropy에서 뺀 것이 Info gain이므로 info gain이 젤 큰 splitting attribute을 고른 것이다.

그렇게 나눈 partition들을 partitions, split_candidates에서 best_attribute을 뺀 리스트를 new_candidates에 담는다.

하위 partitions안에 partition개수 만큼 subtree가 생긴다. 이 subtree는
subtrees = {attribute_value(split한 attribute value1): {subtree}, split한 attribute value2: {subtree}, ...}
로 build 한다.

```
    # 기본
    list = [inputs[k][1] for k in range(num_inputs)]
    leaf = max(list, key=list.count)
    subtrees[None] = leaf
    return (best_attribute, subtrees)
```

Subtreees에서 attribute_value가 없는 경우, 즉 key값이 None인 경우 most voted로 정한다.
(best_attribute, subtrees) 는 예시로 ('student' , {'no': {} , 'yes': {}})의 형태이다.

classify

```
def classify(tree, input):
    if tree in class_values:
        return tree
    # dict
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)


    if str(subtree_key) not in subtree_dict:
        subtree_key = None

    subtree = subtree_dict[subtree_key]
    return classify(subtree, input)
```

```
def classify(tree, input):
    if tree in class_values:
        return tree
```

Input은 classify하려는 data sample 한개이다.

Tree값이 leaf 단계라 class_value값이라면 그 class value로 classify한다.

```
    # dict
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)


    if str(subtree_key) not in subtree_dict:
        subtree_key = None

    subtree = subtree_dict[subtree_key]
    return classify(subtree, input)
```

Attribute로 split한 tree build의 반환값을 attribute, subtree_dict에 담는다.

Subtree_key는 input중에 가장 entropy가 낮아지는 방향으로 input을 split한 attribute value이다. 만약 이러한 값이 없다면 subtree_key는 None으로 설정한다.

그리고 그러한 subtree_key에 해당하는 subtree로 내려가서 다시 leaf, 즉 class value값이 반환될 때 까지 재귀적으로 classify함수를 순회한다.

```
tree = build_tree(inputs)
```

Inputs= argv[1] = train텍스트 파일로 tree를 생성한다.

```
path = argv[2]
f = open(path, "r")
attribute_keys_t = f.readline()
attribute_keys_t = attribute_keys_t.split('\t')
n = len(attribute_keys_t)
attribute_keys_t[n-1] = attribute_keys_t[n-1].strip('\n')
result = []
```

Path = argv[2] = test텍스트 파일이다.

테스트 파일은 class value가 없는 즉 class label이 없는 파일이다.

그러므로 attribute name이 있는 첫째 줄에서 '₩t'을 없애고 리스트에 담고 리스트 마지막 값의 끝에 '₩n'을 없앤 리스트를 attribute_keys_t에 담는다. N개의 test sample이 있고 이러한 n개의 classify 결과를 result 리스트에 담을 것이다.

```
for lines in f:
    str_line = list(lines.split('\t'))
    str_line[n-1] = str_line[n-1].strip('\n')
    Data = dict(zip(attribute_keys_t, str_line))
    result.append(classify(tree, Data))
```

두번째 줄부터는 실제 value들이 주어진다. 똑같이 '₩t', '₩n'을 없애고 str_line 리스트에 담아준다 Attribute_keys_t: str_line 형식의 dict를 Data에 담는다. 이러한 Data하나를 생성한 tree기준으로 classify하고, 그 값을 result 리스트에 추가한다.

```
f = open(argv[2], 'r')
fw = open(argv[3], 'w')
f.readline()
fw.write(keys)
```

처음에 train 텍스트파일의 첫 줄을 keys에 담아뒀다. 이 keys값을 첫 줄에 출력한다.

Attribute value들을 출력하기 위해 test 텍스트 파일을 f에, train 텍스트 파일에 value들과 classify 된 result 값들을 쓰기 위해 argv[3] = result 텍스트 파일을 fw에 담는다.

```
k = 0
for i in range(len(result)):
    lines = f.readline()
    str_line = lines.strip('\n')
    fw.write(str_line+'\t')
    fw.write(result[k])
    fw.write('\n')
    k+=1
```

K는 result리스트를 하나씩 접근할 index 이므로 처음에 0으로 초기화한다.

Result를 다 접근할 때 까지

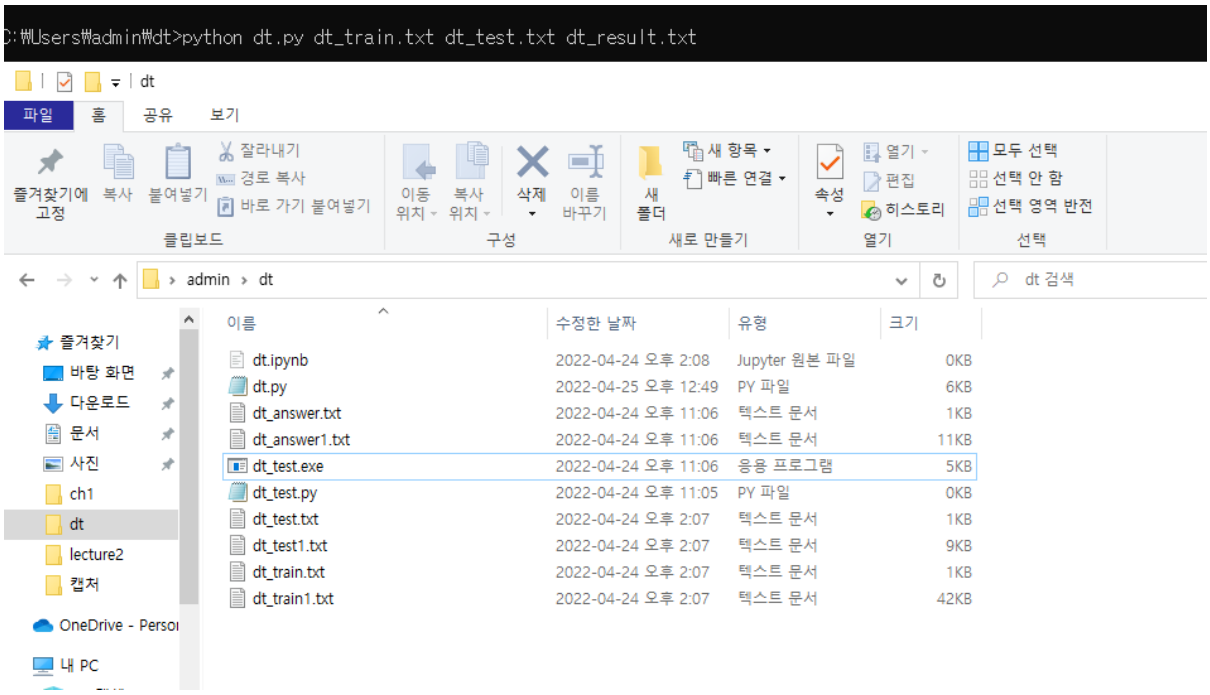
Test텍스트 파일의 data sample들의 class value를 제외한 value값들을 끝에 '₩n'를 빼고 result 텍스트 파일에 쓴다.

이 때, 그 뒤에 class value를 출력해야 하기 때문에 '₩t'를 붙여서 result 텍스트 파일에 쓴다.

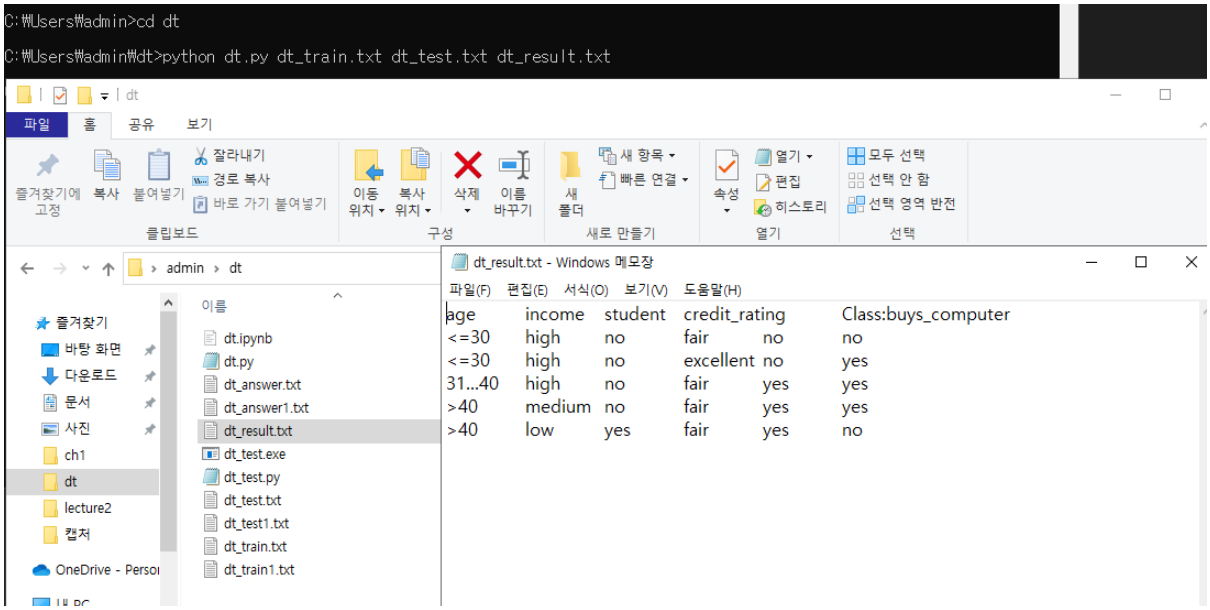
그리고 그에 해당하는 result[k]를 출력하고 줄바꿈한다. K index값을 1씩 증가하면서 모든 test텍스트 파일을 돌면서 result 텍스트 파일에 형식에 맞게 쓴다.

컴파일 방법

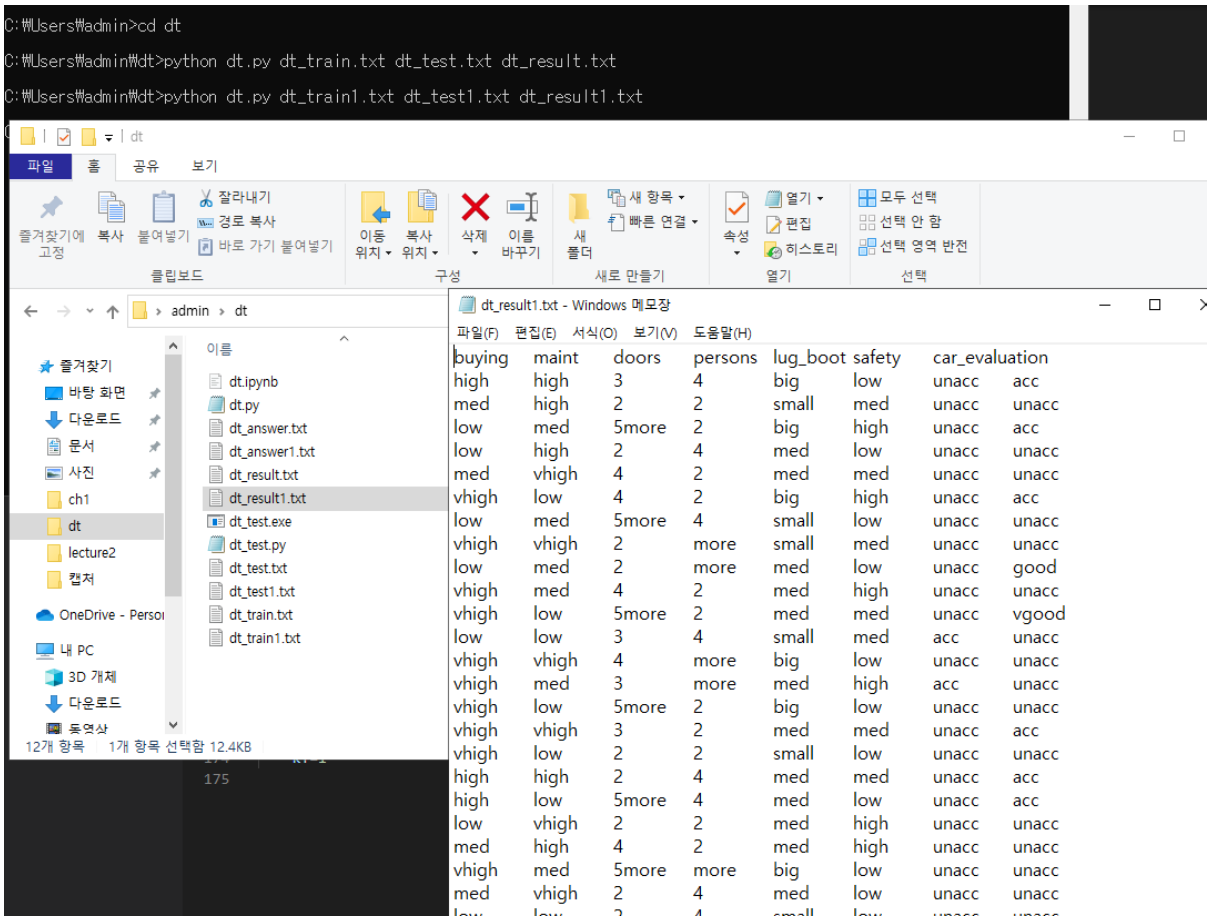
1. 해당 파일 디렉토리 아래 dt.py, .txt파일들을 넣고 python dt.py dt_train.txt dt_result.txt를 입력한다.



2. dt_result.txt 파일이 생겼다.



3. dt_train1.txt에 대해서도 dt_result1.txt를 생성한다.



4. dt_test.exe dt_answer.txt dt_result.txt

dt_test.exe dt_answer1.txt dt_result1.txt

실제 정답 파일과 내 decision tree로 classify한 결과를 비교한다.

```
C:¥Users¥admin¥dt>dt_test.exe dt_answer.txt dt_result.txt
5 / 5

C:¥Users¥admin¥dt>dt_test.exe dt_answer1.txt dt_result1.txt
315 / 346

C:¥Users¥admin¥dt>
```