code

1.DcTree(class)

```
from math import log


class DcTree:

    def __init__(self, file_path):
        """attr为包含的属性 , DataSet为数据集"""
        # 为attr各个属性值中所对应的取值
        # key为 attr value为取值 lists
        self.labels_value = {}
        self.attr = []
        self.DataSet = []
        self.read_data(file_path)
        # 数据集的个数
        self.data_number = len(self.DataSet)
        # 最后的数图
        self.tree_map = {}
        self.TotalGain = self.compute_gain(self.DataSet)

    def spilt_lists(self, data_set, index, value):
        """将data_set的各个列表的index位置值等于value的列表分离出去"""
        subdata_set = []
        for one_data in data_set:
            if one_data[index] == value:
                # 提出该行index位置的元素
                temp_set = one_data[:index]
                temp_set.extend(one_data[index+1:])
                subdata_set.append(temp_set)
        return subdata_set

    def get_best_attr(self, attr, data_set):
        """返回attr中信息熵最大的属性的位置"""
        # print(sub_data_set)
        if len(attr) == 0:
            return
        best_index = current_index = 0
        max_gain = 0
        times = len(attr) - 1
        while current_index < times:
            # 计算the_attr的熵
            the_attr_gain = 0
            for val in self.labels_value[attr[current_index]]:
```

```
                # 注意新的信息熵是求的在这个子数据集下的属性的信息熵，所以每个都要重新
计算
                sub_data_set = self.spilt_lists(data_set, current_index,
val)
                temp_gain = self.compute_gain(sub_data_set)
                the_attr_gain += float(len(sub_data_set))/len(data_set) *
temp_gain
            the_attr_gain = self.TotalGain - the_attr_gain
            # print(attr[current_index]+":"+str(the_attr_gain))
            if max_gain < the_attr_gain:
                # 更换最大gain
                max_gain = the_attr_gain
                best_index = current_index
            current_index += 1
        return best_index

    def build_tree(self):
        copy_attr = self.attr
        self.tree_map = self.create_tree(self.DataSet, copy_attr)
        return self.tree_map

    def create_tree(self, data_set, attr):
        """建树"""
        # 先求出attr中gain最大的item
        # 用best_gain_pos定位
        # print(data_set)
        if len(data_set[0]) == 1:
            # 只剩结果
            return data_set[0][0]
        res_lists = [example[-1] for example in data_set]
        if res_lists.count(res_lists[0]) == len(res_lists):
            # 说明此时所有结果集相同
            return res_lists[0]
        best_gain_pos = self.get_best_attr(attr, data_set)
        the_attr = attr[best_gain_pos]
        my_tree_map = dict()
        current_key = the_attr
        current_val = dict()
        # del(attr[best_gain_pos])
        copy_attr = attr[:best_gain_pos]
        copy_attr.extend(attr[best_gain_pos + 1:])
        for attr_val in self.get_classification(data_set, best_gain_pos):
            current_val[attr_val] =
self.create_tree(self.spilt_lists(data_set, best_gain_pos, attr_val),
copy_attr)
        my_tree_map[current_key] = current_val
        return my_tree_map

    @staticmethod
```

```
    def get_classification(data_set, pos):
        """以list形式返回在data_set中在pos位置的种类"""
        the_class = list()
        for the_data in data_set:
            if the_data[pos] not in the_class:
                the_class.append(the_data[pos])
        return the_class

    def compute_gain(self, DataSet):
        """计算传入的DateSet的最后一个元素的信息熵"""
        # Count为数据集的总个数
        count = len(DataSet)
        decision = {}
        # 统计决策种类的个数
        for TheData in DataSet:
            label = TheData[-1]
            if label not in decision.keys():
                decision[label] = 0
            decision[label] += 1
        the_gain = 0
        for key in decision:
            prob = float(decision[key])/count
            the_gain -= prob * log(prob, 2)
        return the_gain

    def read_data(self, file_path):
        """从txt文本中读取数据"""
        filename = file_path
        read_row = 1
        attr = []
        data_set = []
        with open(filename) as FileObj:
            for line in FileObj:
                if read_row == 1:
                    attr = line.split()
                    # 初始化self.labels_value
                    for the_attr in attr[:-1]:
                        self.labels_value[the_attr] = []
                else:
                    split_line = line.split()
                    index = 0
                    # 最后一列不为属性列
                    times = len(split_line) - 1
                    while index < times:
                        if split_line[index] not in
self.labels_value[attr[index]]:

 self.labels_value[attr[index]].append(split_line[index])
                        index += 1
```

```
                    data_set.append(split_line)
                read_row += 1
        self.attr = attr
        self.DataSet = data_set
```

2.main_for_DT

```
from dc_tree import DcTree

# 读取数据并初始化树
file_path = 'data.txt'
my_tree = DcTree(file_path)


# print(my_tree.TotalGain)
# print(my_tree.DataSet)
# print(my_tree.attr)
# print(my_tree.spilt_lists(my_tree.DataSet, 0, 'High'))

# print(my_tree.labels_value)
print(my_tree.build_tree())
```

3.data.txt

```
Turnover Reissued Dividend Buy
High Yes AboveAverage NO
High No Average YES
High No AboveAverage NO
High No Average YES
Medium Yes AboveAverage YES
Medium Yes Average YES
Medium No BelowAverage YES
Medium No AboveAverage NO
Low Yes Average YES
Low Yes BelowAverage NO
Low Yes BelowAverage NO
Low No AboveAverage YES
Low No BelowAverage YES
Low No BelowAverage YES
```

result:

```
{'Dividend': {'AboveAverage': {'Turnover': {'High': 'NO', 'Medium':
{'Reissued': {'Yes': 'YES', 'No': 'NO'}}, 'Low': 'YES'}}, 'Average': 'YES',
'BelowAverage': {'Reissued': {'No': 'YES', 'Yes': 'NO'}}}}
```