School of Data and Computer Science, Sun Yat-sen University

E10 Decision Tree

18340149 孙新梦

November 18, 2020

目录

1 Datasets

The UCI dataset (`http://archive.ics.uci.edu/ml/index.php`) is the most widely used dataset for machine learning. If you are interested in other datasets in other areas, you can refer to `https://www.zhihu.com/question/63383992/answer/222718972`.

Today's experiment is conducted with the **Adult Data Set** which can be found in `http://archive.ics.uci.edu/ml/datasets/Adult`.

Data Set Characteristics:	Multivariate	**Number of Instances:**	48842	**Area:**	Social
Attribute Characteristics:	Categorical, Integer	**Number of Attributes:**	14	**Date Donated**	1996-05-01
Associated Tasks:	Classification	**Missing Values?**	Yes	**Number of Web Hits:**	1305515

You can also find 3 related files in the current folder, `adult.name` is the description of **Adult Data Set**, `adult.data` is the training set, and `adult.test` is the testing set. There are 14 attributes in this dataset:

```
>50K, <=50K.

1. age: continuous.
2. workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov,
State-gov, Without-pay, Never-worked.
3. fnlwgt: continuous.
4. education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm,
Assoc-voc, 9th, 7th-8th, 12th, Masters, 5. 1st-4th, 10th, Doctorate, 5th-6th,
Preschool.
5. education-num: continuous.
6. marital-status: Married-civ-spouse, Divorced, Never-married, Separated,
Widowed, Married-spouse-absent, Married-AF-spouse.
7. occupation: Tech-support, Craft-repair, Other-service, Sales,
Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct,
```

```
Adm-clerical,Farming-fishing,Transport-moving,Priv-house-serv,Protective-serv,
Armed-Forces.
8. relationship: Wife,Own-child,Husband,Not-in-family,Other-relative,Unmarried.
9. race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
10. sex: Female, Male.
11. capital-gain: continuous.
12. capital-loss: continuous.
13. hours-per-week: continuous.
14. native-country: United-States, Cambodia,England,Puerto-Rico,Canada,Germany,
Outlying-US(Guam-USVI-etc),India,Japan,Greece, South,China,Cuba,Iran,Honduras,
Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France,
Dominican-Republic,Laos,Ecuador,Taiwan, Haiti, Columbia,Hungary,Guatemala,
Nicaragua,Scotland,Thailand,Yugoslavia,El-Salvador, Trinadad&Tobago,Peru,Hong,
Holand-Netherlands.
```

Prediction task is to determine whether a person makes over 50K a year.

2 Decision Tree

2.1 ID3

ID3 (Iterative Dichotomiser 3) was developed in 1986 by Ross Quinlan. The algorithm creates a multiway tree, finding for each node (i.e. in a greedy manner) the categorical feature that will yield the largest information gain for categorical targets. Trees are grown to their maximum size and then a pruning step is usually applied to improve the ability of the tree to generalise to unseen data.

ID3 Algorithm:

1. Begins with the original set S as the root node.
2. Calculate the entropy of every attribute a of the data set S.
3. Partition the set S into subsets using the attribute for which the resulting entropy after splitting is minimized; or, equivalently, information gain is maximum.
4. Make a decision tree node containing that attribute.
5. Recur on subsets using remaining attributes.

Recursion on a subset may stop in one of these cases:

- every element in the subset belongs to the same class; in which case the node is turned into a leaf node and labelled with the class of the examples.
- there are no more attributes to be selected, but the examples still do not belong to the same class. In this case, the node is made a leaf node and labelled with the most common class of the examples in the subset.
- there are no examples in the subset, which happens when no example in the parent set was found to match a specific value of the selected attribute.

ID3 shortcomings:

- ID3 does not guarantee an optimal solution.
- ID3 can overfit the training data.
- ID3 is harder to use on continuous data.

Entropy:

Entropy $H(S)$ is a measure of the amount of uncertainty in the set S.

$$H(S) = \sum_{x \in X} -p(x) \log_2 p(x)$$

where

- S is the current dataset for which entropy is being calculated
- X is the set of classes in S
- $p(x)$ is the proportion of the number of elements in class x to the number of elements in set S.

Information gain:

Information gain $IG(A)$ is the measure of the difference in entropy from before to after the set S is split on an attribute A. In other words, how much uncertainty in S was reduced after splitting set S on attribute A.

$$IG(S, A) = H(S) - \sum_{t \in T} p(t)H(t) = H(S) - H(S \mid A)$$

where

- $H(S)$ is the entropy of set S
- T is the subsets created from splitting set S by attribute A such that $S = \cup_{t \in T} t$
- $p(t)$ is the proportion of the number of elements in t to the number of elements in set S
- $H(t)$ is the entropy of subset t.

2.2 C4.5 and CART

C4.5 is the successor to ID3 and removed the restriction that features must be categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of if-then rules. These accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule’s precondition if the accuracy of the rule improves without it.

C5.0 is Quinlan’s latest version release under a proprietary license. It uses less memory and builds smaller rulesets than C4.5 while being more accurate.

CART (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node.

3 Tasks

- Given the training dataset `adult.data` and the testing dataset `adult.test`, please accomplish the prediction task to determine whether a person makes over 50K a year in `adult.test` by using ID3 (or C4.5, CART) algorithm (C++ or Python), and compute the accuracy.

    1. You can process the continuous data with **bi-partition** method.

2. You can use prepruning or postpruning to avoid the overfitting problem.
3. You can assign probability weights to solve the missing attributes (data) problem.

- Please finish the experimental report named `E10_YourNumber.pdf`, and send it to `ai_2020@foxmail.com`

4 Codes and Results

Code

```
”””
E10 decisionTree.py姓名：孙新梦学号：

18340149
TASK使用:算法实现决策树，判断一个人是否每年能拿到超过万ID35输入：两个文件：

    adult.: 训练集data
    adult.: 测试集test输出：个测试样例的精确度，最后的平均精确度
    10运行：

    python decisionTree.py 普通版 #
    python decisionTree.py —post_proning 1 后剪枝#
    python decisionTree.py —print_tree 1 打印决策树到文件中#
    python decisionTree.py —ignore 1 忽略掉第,，特征#71213
”””
import numpy as np
import random
import copy
import operator
import argparse


class Node(object):
    def __init__(self, data_index, classfy=None, is_leaf=False):
        self.data_index = data_index  # 记录到达该节点的数据的序号
```

```
        self.classfy = classfy  # 如果是叶节点，记录类别（代表0<50k，代表1>=50K）；否则为None
        self.is_leaf = is_leaf  # 判断是否为叶节点
        if is_leaf:
            self.son = None
        else:
            self.son = []  # 记录子节点
        self.divide_feature_index = None  # 该节点用于划分的特征序号
        self.divide_value = None  # 该节点用于划分的特征的值，若是连续变量，则表示小于该值的数据进入该节点
        self.father = None  # 记录节点的父节点
        self.brother_index = None  # 记录节点在兄弟结点中的第几个
        self.miss_index = []  # 记录到达该节点的，在当前划分的特征序号上有缺失值的数据的序号
        self.probability = 1  # 记录该节点的权重（权重不继承）

    def setDivideFeatureAndValue(self, divide_feature_index, divide_value):
        self.divide_feature_index = divide_feature_index
        self.divide_value = divide_value

    def setBrotherIndex(self, index):
        self.brother_index = index

    def setMissIndex(self, miss_index, probability):
        self.miss_index = miss_index
        self.probability = probability

    # 得到一个结点的深拷贝
    def copy(self):
        new_node = Node(self.data_index, self.classfy, self.is_leaf)
        new_node.data_index = copy.deepcopy(self.data_index)
        new_node.setDivideFeatureAndValue(self.
```

```
            divide_feature_index, self.divide_value)
        new_node.setBrotherIndex(self.brother_index)
        new_node.son = []
        if not new_node.is_leaf:
            for i in range(len(self.son)):
                son_copy = self.son[i].copy()
                son_copy.father = new_node
                new_node.son.append(son_copy)
        return new_node

    # 判断两个结点是否相同（用到达该节点的数据的序号来判断）
    def equal(self, next_node):
        return operator.eq(self.data_index, next_node.data_index)

    # 判断两个结点是否完全相同
    def allEqual(self, next_node):
        if self.is_leaf and next_node.is_leaf:
            return self.equal(next_node)
        t = (self.is_leaf == next_node.is_leaf)
        t = t and self.equal(next_node)
        for i in range(len(self.son)):
            t = t and self.son[i].allEqual(next_node.son[i])
        return t

    # 得到以该节点为根的子树中的所有叶子节点
    def getLeaf(self, leaf_list):
        if self.is_leaf:
            leaf_list.append(self)
            return
        for x in self.son:
            x.getLeaf(leaf_list)

    # 将该节点转化为字典
```

```python
    def getNodeDict(self, label_index, feature_name):
        if self.is_leaf:
            return label_index[self.classfy]
        res = {}
        for x in self.son:
            now_feature_name = feature_name[x.divide_feature_index]
            if isinstance(x.divide_value, str):
                res[now_feature_name + ": " + x.divide_value] = x.getNodeDict(label_index, feature_name)
            else:
                res[now_feature_name + ": <" + str(x.divide_value)] = x.getNodeDict(label_index, feature_name)
        return res


class Tree(object):
    def __init__(self, features, labels, max_continous_son, max_leaf_length):
        self.features = features  # 训练feature
        self.labels = labels  # 训练labels
        self.feature_length = len(self.features[0])  # 特征维度
        self.max_continous_son = max_continous_son  # 连续变量划分次数
        self.max_leaf_length = max_leaf_length  # 叶子节点的最大数据量
        self.root = Node(list(range(0, len(features))))  # 树的根节点
        self.leaf_nodes = []  # 记录所有叶子节点
        self.ignore_list = []  # 记录忽略的特征序号

    def setContinuousIndex(self, index_list):
        self.continous_list = index_list

```

```
    def setIgnoreIndex(self, ignore_list):
        self.ignore_list = ignore_list

    def isContinuous(self, index):
        return index in self.continous_list

    # 树的深拷贝
    def treeCopy(self):
        new_tree = Tree(self.features, self.labels, self.
            max_continous_son, self.max_leaf_length)
        new_tree.setContinuousIndex(self.continous_list)
        new_tree.setIgnoreIndex(self.ignore_list)
        new_tree.root = self.root.copy()
        new_tree.leaf_nodes = []
        new_tree.root.getLeaf(new_tree.leaf_nodes)
        return new_tree

    def setMaxSon(self, max_continous_son):
        self.max_continous_son = max_continous_son

    def setMaxLeafLength(self, max_leaf_length):
        self.max_leaf_length = max_leaf_length

    # 清空root
    def deleteRoot(self):
        self.root = Node(list(range(0, len(self.features))))

    # 对数据根据其中一个特征进行划分
    def getSubIndex(self, data_index, feature_index):

        miss_data_indexs = []
        if self.isContinuous(feature_index):
            now_data = [self.features[x][feature_index] for x in
```

```
data_index]
            # 连续变量直接根据最大值和最小值分成个区间max_continous_son
            max_value = max(now_data) + 1
            min_value = min(now_data)
            step = (max_value - min_value) / self.max_continous_son
            sub_data_indexs = [[] for x in range(self.max_continous_son)]  # 记录划分后的数据
            for i in data_index:
                for j in range(0, self.max_continous_son):
                    if self.features[i][feature_index] < min_value + (j + 1) * step:
                        sub_data_indexs[j].append(i)
                        break
            # 去除不必要的划分
            for i in range(self.max_continous_son - 1, 0, -1):
                if len(sub_data_indexs[i]) == 0:
                    sub_data_indexs.pop(i)

            return (min_value, step), sub_data_indexs, miss_data_indexs  # 返回（最小值，步长），划分后的数据，有缺失值的数据

        sub_data_indexs = {}
        for i in data_index:
            if self.features[i][feature_index] == '?':
                miss_data_indexs.append(i)
                continue
            if self.features[i][feature_index] not in sub_data_indexs:
                sub_data_indexs[self.features[i][feature_index]] = [i]
            else:
```

```
                    sub_data_indexs[self.features[i][feature_index]].append(i)
        return None, sub_data_indexs, miss_data_indexs  # 返回，划分后的数据，有缺失值的数据None

    # 将得到的数据根据计算概率label
    def getCount(self, data_index):
        count = {}
        data_len = len(data_index)
        if data_len > 0:
            add_count = 1 / data_len
        for i in data_index:
            if self.labels[i] not in count:
                count[self.labels[i]] = add_count
            else:
                count[self.labels[i]] += add_count
        return count

    # 计算信息熵
    def getEntropy(self, count):
        try:
            if len(count.keys()) == 0:
                return 0
            res = 0
            for x in count:
                res += count[x] * np.log2(count[x])
            return -res
        except Exception as e:
            print(e)

    # 计算信息增益
    def getGain(self, now_root, sub_data_indexs, feature_index, now_entropy):
```

```
        if self.isContinuous(feature_index):
            now_data_len = len(now_root.data_index)
        else:
            now_data_len = 0
            for _, y in sub_data_indexs.items():
                now_data_len += len(y)
        sub_entropy_sum = 0
        x = None
        try:
            if self.isContinuous(feature_index):
                for x in sub_data_indexs:
                    now_count = self.getCount(x)
                    sub_entropy_sum += len(x) / now_data_len * self.getEntropy(now_count)
            else:
                for _, y in sub_data_indexs.items():
                    sub_entropy_sum += len(y) / now_data_len * self.getEntropy(self.getCount(y))
            return now_entropy - sub_entropy_sum
        except Exception as e:
            print("Error")
            print(e)
            input()

    # 计算属性的固有值
    def getIV(self, now_root, sub_data_indexs, feature_index):
        if self.isContinuous(feature_index):
            now_data_len = len(now_root.data_index)
        else:
            now_data_len = 0
            for _, y in sub_data_indexs.items():
                now_data_len += len(y)
        res = 0
```

```
        if self.isContinuous(feature_index):
            for x in sub_data_indexs:
                res += len(x) / now_data_len * np.log2(len(x) / now_data_len)
        else:
            for _, y in sub_data_indexs.items():
                res += len(y) / now_data_len * np.log2(len(y) / now_data_len)
        return -res

    # 计算属性的信息增益率
    def getGainRadio(self, now_root, sub_data_indexs, feature_index, now_entropy):
        iv = self.getIV(now_root, sub_data_indexs, feature_index)
        if iv == 0:
            return -1
        return self.getGain(now_root, sub_data_indexs, feature_index, now_entropy) / iv

    # 设定结点类别
    def setClassfy(self, now_root, now_counts):
        num = -1
        classfy = -1
        for x in now_counts:
            if num < now_counts[x]:
                classfy = x
                num = now_counts[x]
        now_root.classfy = classfy

        # 建立新节点

    def buildNode(self, now_root):
        now_counts = self.getCount(now_root.data_index + now_root.
```

```
miss_index)
        if now_counts == None or len(now_counts) == 0:
            return
        # 当到达一个结点的数据小于等于，该节点不再划分，变为叶节点max_leaf_length
        # 或当到达该节点的数据全都是同一种类别时，该节点也不再划分
        if len(now_root.data_index + now_root.miss_index) <= self.max_leaf_length or len(now_counts) == 1:
            now_root.is_leaf = True
            self.setClassfy(now_root, now_counts)
            self.leaf_nodes.append(now_root)
            return

        # 记录最优划分
        best_sub_data_indexs = None
        best_gain = float('-inf')
        best_divide_feature_index = -1
        best_value = None
        best_miss_index = []
        len_data = len(now_root.data_index + now_root.miss_index)
        now_entropy = self.getEntropy(now_counts)

        for i in range(self.feature_length):
            if i == now_root.divide_feature_index:
                if not self.isContinuous(now_root.divide_feature_index):
                    continue

            if i in self.ignore_list:
                continue

            now_value, now_sub_data_indexs, miss_index = self.getSubIndex(now_root.data_index + now_root.
```

```
                miss_index, i)

            now_gain = (1 - (len(miss_index) / len_data)) * self.
                getGainRadio(now_root, now_sub_data_indexs, i,




            if best_gain < now_gain:
                best_divide_feature_index = i
                best_gain = now_gain
                best_sub_data_indexs = now_sub_data_indexs
                best_value = now_value
                best_miss_index = miss_index

        for i, key in enumerate(best_sub_data_indexs):
            if self.isContinuous(best_divide_feature_index):
                new_son = Node(key)
                new_son.setDivideFeatureAndValue(
                    best_divide_feature_index, best_value[0] +
                    best_value[1] * (i + 1))
            else:
                new_son = Node(best_sub_data_indexs[key])
                new_son.setDivideFeatureAndValue(
                    best_divide_feature_index, key)
            new_son.setMissIndex(best_miss_index, len(new_son.
                data_index) / len_data)

            new_son.father = now_root
            new_son.setBrotherIndex(i)
            self.buildNode(new_son)  # 递归生成树
            now_root.son.append(new_son)

```

```python
    # 训练
    def train(self):
        self.buildNode(self.root)

    # 预测的子函数
    def predictNode(self, now_node, test_feature):
        if now_node.is_leaf:
            return now_node.classfy

        now_divide_feature_index = now_node.son[0].divide_feature_index
        miss = []

        for i in range(len(now_node.son)):
            if self.isContinuous(now_divide_feature_index):
                if test_feature[now_divide_feature_index] < now_node.son[i].divide_value:
                    return self.predictNode(now_node.son[i], test_feature)
                elif i == len(now_node.son) - 1 and test_feature[now_divide_feature_index] >= now_node.son[i].divide_value:
                    return self.predictNode(now_node.son[i], test_feature)
            else:
                miss = miss + [i for _ in now_node.son[i].data_index + now_node.son[i].miss_index]
                if test_feature[now_divide_feature_index] == now_node.son[i].divide_value:
                    return self.predictNode(now_node.son[i], test_feature)

        if not self.isContinuous(now_divide_feature_index):
```

```
            # 根据【到达子节点的数据的长度】决定进入哪一个子节点
            # 例如，父节点总共有条数据，第一个子节点里有条数据，第二个子节点有条数据，第三个子节点有两条数据1035
            # 那么的值为miss[0, 0, 0, 1, 1, 1, 1, 1, 2, 2]
            # 将打乱后，再随机取其中一个元素，这样就可以实现以一定概率进入子节点miss
            random.shuffle(miss)
            son_index = miss[random.randint(0, len(miss) - 1)]
            return self.predictNode(now_node.son[son_index], test_feature)

        # 否则随机返回一个类别
        return self.labels[random.randint(0, len(self.labels) - 1)]

    # 总的预测函数
    def predictAll(self, test_features, test_labels):
        res = 0
        for i, test_feature in enumerate(test_features):
            pred = self.predictNode(self.root, test_feature)
            if pred == test_labels[i]:
                res += 1
        return res / len(test_labels)

    # 预测函数（带细节）
    def predictAllDetail(self, test_features, test_labels, label_index):
        res = 0
        for i, test_feature in enumerate(test_features):
            pred = self.predictNode(self.root, test_feature)
            print("Pred: %s, Label: %s" % (label_index[pred], label_index[test_labels[i]]), end=" ")
            if pred == test_labels[i]:
                print("True")
```

```
                res += 1
            else:
                print("False")
        print("Accuracy: %.4f" % (res / len(test_labels)))

    # 将树根转化为字典
    def getTreeDict(self, label_index, feature_name):
        return self.root.getNodeDict(label_index, feature_name)


# read data
def readData(train_data, train_label, test_data, test_label, label_dict):
    def isContinuous(i):
        return i in [0, 2, 4, 10, 11, 12]

    # read training data
    with open("adult.data", "r", encoding='utf8') as f:
        j = 0
        for line in f.readlines():
            try:
                nowData = str(line)[:-1].replace(' ', '').split(',')
                if nowData[-1] == '':
                    continue
                train_label[j] = label_dict[nowData[-1]]
                for i, data in enumerate(nowData[:-1]):
                    if isContinuous(i):
                        train_data[j].append(int(data))
                    else:
                        train_data[j].append(data)
                j += 1
            except Exception as e:
```

```
                print("Error occured in line %d" % j)
                print(e)

    # read testing data
    with open("adult.test", "r", encoding='utf8') as f:
        j = 0
        flag = 0
        for line in f.readlines():
            try:
                if flag == 0:
                    flag = 1
                    continue
                nowData = str(line)[:-2].replace(' ', '').split
                    (',')
                if nowData[-1] == '':
                    continue
                test_label[j] = label_dict[nowData[-1]]
                for i, data in enumerate(nowData[:-1]):
                    if isContinuous(i):
                        test_data[j].append(int(data))
                    else:
                        test_data[j].append(data)
                j += 1
            except Exception as e:
                print("Error occured in line %d" % j)
                print(e)


# 调整超参数以达到最优正确率（耗时很长）
def getBestParameter(train_data, train_label, test_data,
    test_label, continous_list):
    best_max_continous_son = 10
    best_max_leaf_length = 8
```

```python
    best_rate = 0

    tree = Tree(train_data, train_label, 2, 2)
    tree.setContinuousIndex(continous_list)
    son_begin = 2
    son_end = 6
    length_begin = 30
    length_end = 51
    for max_continous_son in range(son_begin, son_end):
        for max_leaf_length in range(length_begin, length_end):
            tree.setMaxSon(max_continous_son)
            tree.setMaxLeafLength(max_leaf_length)
            print("Max Continous Son: %d, Max Leaf Length: %d" % (max_continous_son, max_leaf_length))
            print("Train ...")
            tree.train()
            print("Done ...")

            print("Predict ...")
            rate = tree.predictAll(test_data, test_label)
            print("Accuracy: %.4f" % (rate))
            print()
            if best_rate < rate:
                best_rate = rate
                best_max_continous_son = max_continous_son
                best_max_leaf_length = max_leaf_length

            tree.deleteRoot()
    return best_max_continous_son, best_max_leaf_length


# 后剪枝（耗时很长）
# 一个贪心的剪枝，如果某个剪枝能得到好的效果，那么会基于新生成的树接着剪枝
```

```
def postPruningFast(base_tree, validation_features,
    validation_labels, break_points=10000):
    first_rate = base_tree.predictAll(validation_features,
        validation_labels)

    is_vis = {}
    best_rate = first_rate
    best_tree = base_tree.treeCopy()

    out_index = 0
    index = 0

    while out_index + index < break_points:
        now_base = best_tree.treeCopy()
        is_vis = {}

        len_leaf = len(now_base.leaf_nodes)
        index = 0
        flag = 0
        step = 0
        while index < len_leaf:
            print(out_index + index, "Best rate:", best_rate)
            leaf = now_base.leaf_nodes[index]
            if leaf.father == None:
                index += 1
                continue
            key_tuple = tuple(leaf.father.data_index)
            if key_tuple in is_vis:
                index += 1
                continue
            now_father = leaf.father.father
            if now_father == None:
                index += 1
```

```python
                continue

            temp = now_base.treeCopy()
            brother_index = leaf.father.brother_index
            now_count = now_base.getCount(leaf.father.data_index)

            now_father.son[brother_index].is_leaf = True
            now_base.setClassfy(now_father.son[brother_index], now_count)

            now_rate = now_base.predictAll(validation_features, validation_labels)
            # 减少预测时的随机操作对正确率的影响
            # 随机会导致正确率波动，只有当新的正确率比原本的正确率高于一个阈值，才能说明剪枝正确
            if now_rate - best_rate > 0.0007 - 0.00005 * step:
                flag = 1
                new_leaf_nodes = now_base.leaf_nodes
                for son_node in leaf.father.son:
                    new_leaf_nodes = list(filter(lambda x: not x.equal(son_node), new_leaf_nodes))
                new_leaf_nodes.append(now_father.son[brother_index])
                now_base.leaf_nodes = new_leaf_nodes
                now_father.son[brother_index].son = []
                best_rate = now_rate
                best_tree = now_base.treeCopy()
                break

            is_vis[key_tuple] = 1
            now_base = temp
            # 最多一万次，节省时间
            if out_index + index > break_points:
```

```
                    return best_tree
                index += 1

            # 如果对当前的树没有剪枝，那么不再继续
            if not flag:
                break
            out_index += index
            step += 1

    return best_tree.treeCopy()


# 由于预测有随机的部分，所以需要预测次求平均值10
def testTen(base_tree, test_data, test_label):
    print("Test 10 times...\n")
    rate_10 = 0
    rate = 0
    for i in range(10):
        print("Predict %d ..." % i)
        rate = base_tree.predictAll(test_data, test_label)
        print("Accuracy: %.4f" % rate)
        rate_10 += rate
        print()

    rate_10 /= 10
    print("Total Accuracy:", rate_10)


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--best_parameter', type=int, default=0,
                              help若值为，进行自动调参='1')
    parser.add_argument('--post_pruning', type=int, default=0,
```

```
                        help若值为，进行后剪枝='1')
    parser.add_argument('--print_tree', type=int, default=0,
                        help若值为，生成树的='1文件json')
    parser.add_argument('--ignore', type=int, default=0,
                        help若值为，忽略第、、个特征='171213')
    args = parser.parse_args()

    train_data = [[] for _ in range(32561)]
    train_label = [0 for _ in range(32561)]
    test_data = [[] for _ in range(16281)]
    test_label = [[] for _ in range(16281)]
    label_dict = {'<=50K': 0, '>50K': 1}
    label_index = ['<=50K', '>50K']
    continous_list = [0, 2, 4, 10, 11, 12]
    ignore_list = []

    print("Read Data ...")
    readData(train_data, train_label, test_data, test_label,
        label_dict)
    print("Done ...")

    # 是否忽略
    if args.ignore == 1:
        ignore_list = [7, 12, 13]

    best_max_continous_son, best_max_leaf_length = 3, 35
    # 自动调参（耗时小时左右）1
    if args.best_parameter == 1:
        best_max_continous_son, best_max_leaf_length =
            getBestParameter(train_data, train_label, test_data,
            test_label,
                                                                continou
                                                                  )
```

```python
        print("Best Max Continous Son:", best_max_continous_son)
        print("Best Max Leaf Length:", best_max_leaf_length)

    best_tree = Tree(train_data, train_label, best_max_continous_son, best_max_leaf_length)
    best_tree.setContinuousIndex(continous_list)
    best_tree.setIgnoreIndex(ignore_list)
    print("Train ...")
    best_tree.train()
    print("Done ...")

    # 后剪枝（由于每次选取的验证集不同，每次剪枝的效果也会不同）
    if args.post_pruning == 1:
        validate_index = list(range(16281))
        # 随机选择验证集
        random.shuffle(validate_index)
        validate_index = validate_index[:3281]
        validation_features = [test_data[i] for i in validate_index]
        validation_labels = [test_label[i] for i in validate_index]
        print("Post pruning ...")
        # 耗时小时1-2
        best_tree = postPruningFast(best_tree, validation_features, validation_labels)
        print("Done ...")

    # 打印树
    if args.print_tree == 1:
        import json

        feature_name = ['age', 'workclass', 'fnlwgt', 'education',
```

```
'education-num',
                    'marital-status', 'occupation', '
                        relationship', 'race', 'sex',
                    'capital-gain', 'capital-loss', 'hours-per
                        -week', 'native-country']
        print("Print Tree ...")
        with open("Tree.json", "w") as f:
            json.dump(best_tree.getTreeDict(label_index,
                feature_name), f)
        print("Done ...")

    testTen(best_tree, test_data, test_label)
```

Result

```
(.venv) D:\学校文件\上课\大三上\人工智能实验\平时实验\E10_20201116_DT\E10_20201116_DT>python 参考2.p
y --print_tree 1
Read Data ...
Done ...
Train ...
Done ...
Print Tree ...
Done ...
Test 10 times...

Predict 0 ...
Accuracy: 0.8509

Predict 1 ...
Accuracy: 0.8516

Predict 2 ...
Accuracy: 0.8507

Predict 3 ...
Accuracy: 0.8509
```

```
Predict 4 ...
Accuracy: 0.8511

Predict 5 ...
Accuracy: 0.8510

Predict 6 ...
Accuracy: 0.8514

Predict 7 ...
Accuracy: 0.8504

Predict 8 ...
Accuracy: 0.8514
```

```
Predict 9 ...
Accuracy: 0.8511

Total Accuracy: 0.851034948713224

(.venv) D:\学校文件\上课\大三上\人工智能实验\平时实验\E10_20201116_DT\E10_20201116_DT>
```

5 感想体会

本场实验花费了很多时间，感到自己的python编程基础不大牢靠，并且对于很多数据结构的实现都很没有概念。在网上参考较多，希望之后能做的越来越好。决策树算法的思想是这样的，通过每次选取信息增益最大的属性（也就是用这个属性来划分数据集能够让期待的label分的越开越好），来划分数据集，建立决策树，能够输入一个数据的属性向量后通过判断分支来决定label，之后用测试集验证分类的准确度。