从零开始用Python构造决策树（附公式、代码）

本文介绍如何不利用第三方库，仅用python自带的标准库来构造一个决策树。

起步

熵的计算:

根据计算公式:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 (p_i)$$

对应的 python 代码:

```python
import math
import collections

def entropy(rows: list) -> float:
    """
    计算数组的熵
    """
    result = collections.Counter()
    result.update(rows)
    rows_len = len(rows)
    assert rows_len    # 数组长度不能为0
    # 开始计算熵值
    ent = 0.0
    for r in result.values():
        p = float(r) / rows_len
        ent -= p * math.log2(p)
    return ent
```

条件熵的计算:

根据计算方法：

$$H(Y|X) = \sum_{i=1}^{n} p_i H(Y|X = x_i)$$

对应的 python 代码:

```python
def condition_entropy(future_list: list, result_list: list) -> float:
    """
    计算条件熵
    """
    entropy_dict = collections.defaultdict(list)  # {0:[], 1:[]}
    for future, value in zip(future_list, result_list):
        entropy_dict[future].append(value)
    # 计算条件熵
    ent = 0.0
    future_len = len(future_list)  # 数据个数
    for value in entropy_dict.values():
        p = len(value) / future_len * entropy(value)
        ent += p

    return ent
```

其中参数 future_list 是某一特征向量组成的列表，result_list 是 label 列表。

信息增益：

根据信息增益的计算方法：

$$gain(A) = H(D) - H(D|A)$$

对应的python代码:

```python
def gain(future_list: list, result_list: list) -> float:
    """
    获取某特征的信息增益
    """
    info = entropy(result_list)
    info_condition = condition_entropy(future_list, result_list)
    return info - info_condition
```

定义决策树的节点

作为树的节点，要有左子树和右子树是必不可少的，除此之外还需要其他信息:

```python
class DecisionNode(object):
    """
    决策树的节点
    """
    def __init__(self, col=-1, data_set=None, labels=None, results=None, t
b=None, fb=None):
        self.has_calc_index = []    # 已经计算过的特征索引
        self.col = col              # col 是待检验的判断条件，对应列索引值
        self.data_set = data_set    # 节点的 待检测数据
        self.labels = labels        # 对应当前列必须匹配的值
        self.results = results      # 保存的是针对当前分支的结果，有值则表示该
点是叶子节点
        self.tb = tb                # 当信息增益最高的特征为True时的子树
        self.fb = fb                # 当信息增益最高的特征为False时的子树
```

树的节点会有两种状态，叶子节点中 results 属性将保持当前的分类结果。非叶子节点中，col 保存着该节点计算的特征索引，根据这个索引来创建左右子树。

has_calc_index 属性表示在到达此节点时，已经计算过的特征索引。特征索引的数据集上表现是列的形式，如数据集（不包含结果集）：

```
[
    [1, 0, 1],
    [0, 1, 1],
    [0, 0, 1],
]
```

有三条数据，三个特征，那么第一个特征对应了第一列 [1, 0, 0] ， 它的索引是 0 。

递归的停止条件

本章将构造出完整的决策树，所以递归的停止条件是所有待分析的训练集都属于同一类:

```
def if_split_end(result_list: list) -> bool:
    """
    递归的结束条件，每个分支的结果集都是相同的分类
    """
    result = collections.Counter(result_list)
    return len(result) == 1
```

从训练集中筛选最佳的特征:

```
def choose_best_future(data_set: list, labels: list, ignore_index: list) -
> int:
    """
    从特征向量中筛选出最好的特征，返回它的特征索引
    """
    result_dict = {}  # { 索引: 信息增益值 }
    future_num = len(data_set[0])
    for i in range(future_num):
        if i in ignore_index: # 如果已经计算过了
            continue
        future_list = [x[i] for x in data_set]
        result_dict[i] = gain(future_list, labels) # 获取信息增益
    # 排序后选择第一个
    ret = sorted(result_dict.items(), key=lambda x: x[1], reverse=True)
    return ret[0][0]
```

因此计算节点就是调用 best_index = choose_best_future(node.data_set, node.labels, node.has_calc_index) 来获取最佳的信息增益的特征索引。

构造决策树

决策树中需要一个属性来指向树的根节点，以及特征数量。不需要保存训练集和结果集，因为这部分信息是保存在树的节点中的。

```
class DecisionTreeClass():
    def __init__(self):
        self.future_num = 0      # 特征
        self.tree_root = None    # 决策树根节点
```

创建决策树：

这里需要递归来创建决策树：

```
def build_tree(self, node: DecisionNode):
    # 递归条件结束
    if if_split_end(node.labels):
        node.results = node.labels[0] # 表明是叶子节点
        return
    #print(node.data_set)
    # 不是叶子节点，开始创建分支
    best_index = choose_best_future(node.data_set, node.labels, node.has_c
alc_index)
    node.col = best_index

    # 根据信息增益最大进行划分
    # 左子树
    tb_index = [i for i, value in enumerate(node.data_set) if value[best_i
ndex]]
    tb_data_set     = [node.data_set[x] for x in tb_index]
    tb_data_labels  = [node.labels[x] for x in tb_index]
    tb_node = DecisionNode(data_set=tb_data_set, labels=tb_data_labels)
    tb_node.has_calc_index = list(node.has_calc_index)
    tb_node.has_calc_index.append(best_index)
    node.tb = tb_node
```

```
    # 右子树
    fb_index = [i for i, value in enumerate(node.data_set) if not value[be
st_index]]
    fb_data_set = [node.data_set[x] for x in fb_index]
    fb_data_labels = [node.labels[x] for x in fb_index]
    fb_node = DecisionNode(data_set=fb_data_set, labels=fb_data_labels)
    fb_node.has_calc_index = list(node.has_calc_index)
    fb_node.has_calc_index.append(best_index)
    node.fb = fb_node

    # 递归创建子树
    if tb_index:
        self.build_tree(node.tb)
    if fb_index:
        self.build_tree(node.fb)
```

根据信息增益的特征索引将训练集再划分为左右两个子树。

训练函数

也就是要有一个 fit 函数:

```
def fit(self, x: list, y: list):
    """
    x是训练集，二维数组。y是结果集，一维数组
    """
    self.future_num = len(x[0])
    self.tree_root = DecisionNode(data_set=x, labels=y)
    self.build_tree(self.tree_root)
    self.clear_tree_example_data(self.tree_root)
```

清理训练集

训练后，树节点中数据集和结果集等就没必要的，该模型只要 col 和 result 就可以了：

```
def clear_tree_example_data(self, node: DecisionNode):
    """
    清理tree的训练数据
    """
    del node.has_calc_index
    del node.labels
    del node.data_set
    if node.tb:
        self.clear_tree_example_data(node.tb)
    if node.fb:
        self.clear_tree_example_data(node.fb)
```

预测函数

提供一个预测函数:

```
def _predict(self, data_test: list, node: DecisionNode):
    if node.results:
        return node.results
    col = node.col
    if data_test[col]:
        return self._predict(data_test, node.tb)
    else:
        return self._predict(data_test, node.fb)

def predict(self, data_test):
    """
    预测
    """
    return self._predict(data_test, self.tree_root)
```

测试

数据集使用前面《应用篇》中的向量化的训练集:

```python
if __name__ == "__main__":
    dummy_x = [
        [0, 0, 1, 0, 1, 1, 0, 0, 1, 0, ],
        [0, 0, 1, 1, 0, 1, 0, 0, 1, 0, ],
        [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, ],
        [0, 1, 0, 0, 1, 0, 0, 1, 1, 0, ],
        [0, 1, 0, 0, 1, 0, 1, 0, 0, 1, ],
        [0, 1, 0, 1, 0, 0, 1, 0, 0, 1, ],
        [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, ],
        [0, 0, 1, 0, 1, 0, 0, 1, 1, 0, ],
        [0, 0, 1, 0, 1, 0, 1, 0, 0, 1, ],
        [0, 1, 0, 0, 1, 0, 0, 1, 0, 1, ],
        [0, 0, 1, 1, 0, 0, 0, 1, 0, 1, ],
        [1, 0, 0, 1, 0, 0, 0, 1, 1, 0, ],
        [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, ],
        [0, 1, 0, 1, 0, 0, 0, 1, 1, 0, ],
    ]
    dummy_y = [0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]

    tree = DecisionTreeClass()
    tree.fit(dummy_x, dummy_y)

    test_row = [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, ]
    print(tree.predict(test_row))  # output: 1
```