决策树

1 决策树

假设训练集为

$$D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$$

其中，$x_i = (x_i^1, x_i^2, ..., x_i^n)^T$，n 为特征个数；$y_i \in \{1, 2, ..., K\}, i = 1, 2, ..., N$。

决策树每次要从特征 $x^1, x^2, ..., x^n$ 中选出一个特征，通过 if-else 判断语句将样本分类，使样本尽可能的分开。更一般地，决策树算法是递归地选择最优特征，并根据该特征对训练数据进行分割，使得对各个子数据集有一个最好的分类过程。

1.1 特征选择

定义 1：假设随机变量 X 的概率分布为 $P(X = x_i) = p_i, i = 1, 2, ..., n$，则随机变量 X 的**熵**定义为：

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

熵可以表现出随机变量的**不确定性**，熵越大，随机变量的不确定性就越大。

定义 2： **条件熵**$H(Y|X)$ 表示在已知随机变量 X 的条件下随机变量 Y 的不确定性。其定义为：

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

在计算熵时需要**注意**：

1. 通过样本计算的是经验熵和经验条件熵
2. 若有 0 概率，我们定义 $0 \log_2 0 = 0$

定义 3： **信息增益**表示得知特征 X 的信息后而使得类 Y 的信息的不确定性减少的程度。特征 X 对变量 Y 的信息增益 $g(Y,X)$ 定义为 Y 的经验熵 $H(Y)$ 和给定特征 X 条件下 Y 的条件经验熵 $H(Y|X)$ 之差，即：

$$g(Y,X) = H(Y) - H(Y|X)$$

易于发现，**信息增益大的特征具有更强的分类能力**。因此，基于信息增益的特征选择方法是每次计算所有特征的信息增益，并选择信息增益最大的特征。

定义 4： 特征 X 对变量 Y 的**信息增益比**$g_R(Y,X)$ 定义为其信息增益 $g(Y,X)$ 和Y **关于特征** X **的熵** $H_X(Y)$ 之比，即：

$$g_R(Y,X) = \frac{g(Y,X)}{H_X(Y)}$$

其中，$H_X(Y) = \sum_{i \in value(X)} \frac{|Y_i|}{|Y|} \log_2 \frac{|Y_i|}{|Y|}$，$n$ 为特征 X 的取值个数，$|Y_i|$ 为 Y 中其 X 取值为 x_i 的个数。

1.2 ID3 算法

ID3 算法在决策树的各个节点上**应用信息增益准则**选取特征，递归地构建决策树。

设 D 为训练数据。其有 K 个类 $C_k, k = 1,2,...,K, |C_k|$ 为属于类 C_k 的个数，$\sum_{i=1}^{K} |C_k| = |D|$。设特征 A 有 n 个不同的取值 $\{a_1, a_2, ..., a_n\}$，可以根据 A 的取值将 D 分为 n 个子集 $D_1, D_2, ..., D_n$，有 $\sum_{i=1}^{n} |D_i| = |D|$。记 D_{ik} 为集合 D_i 中属于 C_k 的样本，即 $D_{ik} = D_i \cap C_k$。

此时，特征 A 对数据集 D 的信息增益 $g(D,A)$ 为：

$$\begin{aligned} g(D,A) &= H(D) - H(D|A) \\ &= -\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|} - \sum_{i=1}^{n} \frac{|D_i|}{|D|} H(D_i) \\ &= -\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|} + \sum_{i=1}^{n} \frac{|D_i|}{|D|} \sum_{k=1}^{K} \frac{|D_{ik}|}{|D_k|} \log_2 \frac{|D_{ik}|}{|D_k|} \end{aligned}$$

在 ID3 算法中，需要多次计算信息增益。该算法具体如下：

输入： 训练数据集 D，特征集 A 阈值 ϵ。

1. 若 D 中所有实例属于同一类 C_k，则 T 为单节点树，并将 C_k 作为该节点的类标记
2. 若 $A = \emptyset$，则 T 为单节点树，并将 D 中实例数最大的类 C_k 作为该节点的类标记
3. 否则，对 A 中的各特征计算其**信息增益**，选择信息增益最大的特征 A_g

4. 如果特征 A_g 的信息增益小于 ϵ，则设置 T 为单节点树，并将该节点中实例数最大的类 C_k 作为该节点的类标记
5. 否则，对于 A_g 的每一可能值 a_i，将该节点的数据集分割为子集 D_i，建立子树 T_i。若 $D_i = \emptyset$，则将 D_i 的标记设置为该节点的标记，否则将 D_i 中实例数最大的类标记为 D_i 的类
6. 对第 i 个子节点，以 D_i 为训练集，$A - \{A_g\}$ 为特征集，**递归调用** 1-5

输出：决策树 T

1.3 C4.5 算法

C4.5 算法与 ID3 算法十分类似，唯一的区别是 C4.5 算法使用信息增益比进行特征选择。特征 A 对数据集 D 的信息增益比 $g_R(D, A)$ 为：

$$\begin{aligned} g_R(D, A) &= \frac{g(D, A)}{H_A(D)} \\ &= \frac{-\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|} + \sum_{i=1}^{n} \frac{|D_i|}{|D|} \sum_{k=1}^{K} \frac{|D_{ik}|}{|D_k|} \log_2 \frac{|D_{ik}|}{|D_k|}}{-\sum_{i=1}^{n} \frac{|D_i|}{|D|} \log_2 \frac{|D_i|}{|D|}} \end{aligned}$$

其算法如下：

输入：训练数据集 D，特征集 A 阈值 ϵ。

1. 若 D 中所有实例属于同一类 C_k，则 T 为单节点树，并将 C_k 作为该节点的类标记
2. 若 $A = \emptyset$，则 T 为单节点树，并将 D 中实例数最大的类 C_k 作为该节点的类标记
3. 否则，对 A 中的各特征计算其**信息增益比**，选择信息增益最大的特征 A_g
4. 如果特征 A_g 的信息增益小于 ϵ，则设置 T 为单节点树，并将该节点中实例数最大的类 C_k 作为该节点的类标记
5. 否则，对于 A_g 的每一可能值 a_i，将该节点的数据集分割为子集 D_i，建立子树 T_i。若 $D_i = \emptyset$，则将 D_i 的标记设置为该节点的标记，否则将 D_i 中实例数最大的类标记为 D_i 的类
6. 对第 i 个子节点，以 D_i 为训练集，$A - \{A_g\}$ 为特征集，**递归调用** 1-5

输出：决策树 T

1.4 剪枝

决策树无休止的生长会导致树的过拟合，因此需要简化生成的决策树，可对其进行**剪枝**。一种剪枝方式是将叶节点的个数以正则项加入到树的损失函数中。

假设决策树 T 的**叶节点个数**为 $|T|$。t 为树 T 的一个叶节点，该节点有 N_t 个样本点，其中 k 类的样本点有 N_{tk} 个,$k = 1, 2, ..., |T|$。记 $H_t(T)$ 为叶节点 t 上的经验熵，α 为一超参数，则可定义决策树的损失函数为：

$$
\begin{aligned}
C_\alpha(T) &= \sum_{t=1}^{|T|} N_t H_t(T) + \alpha|T| \\
&= \sum_{t=1}^{|T|} N_t(-\sum_{k=1}^{K} \frac{N_{tk}}{N_t} \log_2 \frac{N_{tk}}{N_t}) + \alpha|T| \\
&:= C(T) + \alpha|T|
\end{aligned}
$$

其中 $C(T)$ 相当于模型对数据的**训练误差**，$|T|$ 表示模型的复杂程度。通过 α 对模型的泛化能力进行调节。

在剪枝时，可以通过自下而上的方法递归实现，具体算法如下：

输入：决策树 T，超参数 α

1. 计算所有节点的经验熵
2. 递归地向上回缩。设一组叶节点回缩到父节点前后树分别为 T_B 和 T_A，如果 $C_\alpha(T_A) \leq C_\alpha(T_B)$，则进行剪枝。更一般地，假设该父节点为 P，其子节点分别为 $P_1, P_2, ..., P_n$，节点上的样本个数分别为 $N_1, N_2, ..., N_n$，且 $N = \sum_{i=1}^{n} N_i$。若 $N \cdot H(P) - \sum_{i=1}^{n} N_i H(P_i) \leq \alpha(n-1)$，则进行剪枝
3. 重复 2，直至不能继续

输出：修剪后的子树 T_α

1.5 CART 算法

与 ID3 算法和 C4.5 算法不同，CART 算法假定决策树是**二叉树**，且 CART 算法**既可以用于分类也可以用于回归**。当其用于回归时，使用**平方误差最小化**准则；当其用于分类时，使用**基尼指数最小化**准则。

1.5.1 回归树

决策树相当于将输入空间划分为多个单元，并将新样本分配到其中的某个单元上，将单元的输出值作为新样本的输出结果。假设输入空间被划分成了 M 个单元 $R_1, R_2, ..., R_M$，且在 R_m 上的输出值为 c_m，则回归树模型可表示为：

$$
f(x) = \sum_{m=1}^{M} c_m I(x \in R_m)
$$

对于空间 R_m 上 c_m 的取值，根据平方误差最小化准则，可以得出 c_m 的最优值为 $\hat{c}_m = mean(y_i | x_i \in R_m)$。

对于决策树中节点的划分问题，相当于寻找切分变量 x^j 和切分点 s。两者会将空间切分成两个区域 $R_1(j,s) = \{x | x^j \leq s\}$ 和 $R_2(j,s) = \{x | x^j > s\}$。CART 算法通过最小化以下变量来确定 x^j 和 s:

$$\min_{j,s}[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{x_i\in R_2(j,s)}(y_i-c_2)^2]$$

对于固定的 j,s，代入 $c_1=mean(y_i|x_i\in R_1(j,s))$ 和 $c_2=mean(y_i|x_i\in R_2(j,s))$，可以得到最优的 j,s。不断地对节点进行该规则的划分即可得到一棵 (最小二乘) 回归树。其算法如下：

输入：训练集数据 D。

1. 遍历切分变量 j 和切分点 s，求解

$$\min_{j,s}[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{x_i\in R_2(j,s)}(y_i-c_2)^2]$$

2. 使用 j,s 将节点划分为 $R_1(j,s)=\{x|x^j\leq s\}$ 和 $R_2(j,s)=\{x|x^j>s\}$，并设置 $c_i=\frac{1}{|R_m(j,s)|}\sum_{x\in R_m(j,s)}y_1, m=1,2$
3. 对两个子区域**递归**调用 1,2，直至满足停止条件
4. 生成决策树 $f(x)=\sum_{m=1}^{M}c_mI(x\in R_m)$

输出：回归树 $f(x)$。

1.5.2 分类树

分类树则使用基尼指数选择最优特征和其切分点。

定义 5：假设随机变量 X 的概率分布为 $P(X=x_k)=p_k, k=1,2,...,K$，则随机变量 X 的**基尼指数**定义为：

$$Gini(X)=\sum_{k=1}^{K}p_k(1-p_k)=1-\sum_{k=1}^{K}p_k^2$$

基尼指数越大，样本集合的不确定性也就越大。对于数据集 D，其基尼指数为 $Gini(D)=1-\sum_{k=1}^{K}(\frac{|C_k|}{|D|})^2$。

定义 6：如果根据特征 A 按 $A=a$ 将 D 划分为两个部分 $D_1=x\in D|A(x)=a$ 和 $D_2=x\in D|A(x)\neq a$，则在特征 A 条件下的集合 D 的基尼指数 $Gini(D,A)$ 定义为：

$$Gini(D,A)=\frac{|D_1|}{|D|}Gini(D_1)+\frac{|D_2|}{|D|}Gini(D_2)$$

使用这一指数可以得到 CART 算法：

输入：训练数据 D 和停止条件。

1. 遍历所有特征 A 和其可能的取值 a，根据条件 $A=a$ 将样本空间分为两部分，计算基尼指数 $Gini(D,A=a)$

2. 选择最小的基尼指数对应的特征 A 和对应的切分点 a 对数据集进行切分，生成两个子节点
3. 对两个子节点**递归**调用 1,2，直到满足停止条件
4. 生成 CART 决策树

输出：CART 决策树。

1.5.3 剪枝

对于任意树 T，其剪枝时的损失函数为 $C_\alpha(T)=C(T)+\alpha|T|$。$\alpha$ 的取值决定了树的复杂程度。

对于树中的任意一个节点 t，假设其子树为 T_t。以 t 为单节点的树的损失函数为 $C_\alpha(t)=C(t)+\alpha$，而以 t 为根节点的子树 T_t 的损失函数为 $C_\alpha(T_t)=C(T_t)+\alpha|T_t|$。

显然，当 $\alpha\to 0^+$ 时，$C_\alpha(T_t)<C_\alpha(t)$；当 $\alpha\to+\infty$ 时，$C_\alpha(T_t)>C_\alpha(t)$。因此存在某一 α，使得 $C_\alpha(T_t)=C_\alpha(t)$。事实上，取 $\alpha=\frac{C(t)-C(T_t)}{|T_t|-1}$ 时即可满足条件。在这种情况下，**单节点 t 和以 t 为根节点的树 T_t 拥有相同的损失**，我们倾向于选择更简单的树 t。

特别地，定义 $g(t)=\frac{C(t)-C(T_t)}{|T_t|-1}$。$g(t)$ 的大小就表示了对节点 t 剪枝后整体损失函数的减小程度。显然 $g(t)$ 越小越好，所以对于任意树 T，我们的目标即为找到 T 中 $g(t)$ 最小的节点 t。

此外我们可以证明：存在 $0=\alpha_0<\alpha_1<...<\alpha_n<+\infty$，当 $\alpha\in[\alpha_i,\alpha_{i+1}),i=0,1,...,n$，剪枝得到的子树序列 $\{T_0,T_1,...,T_n\}$ 是嵌套的。这说明：在计算 $g(t)$ 时，我们无需对每个节点单独计算，而是可以自上而下的进行计算。我们可以从根节点开始向下计算，得到最优的剪枝子树列 $\{T_0,T_1,...,T_n\}$ 以及对应的参数 $\{\alpha_0,\alpha_1,...,\alpha_n\}$。随后我们便可以通过交叉验证法从 $\{T_0,T_1,...,T_n\}$ 选出最优的决策树 T_α。注意，此处 α 不是超参数，而是通过交叉验证法自我习得的。

更一般地，CART 剪枝算法如下：

输入：CART 算法生成的决策树 T_0。

1. 记 $k=0,T=T_0,\alpha=+\infty$
2. 自上而下逐层对决策树中的每个节点 t 计算 $C(T_t),|T_t|,g(t)=\frac{C(t)-C(T_t)}{|T_t|-1}$ 和 $\alpha=\min(\alpha,g(t))$，其中 $C(T_t)$ 是对训练数据的预测误差
3. 对 $g(t)=\alpha$ 的内部节点进行剪枝，对节点 t 以多数表决法决定其类，得到树 T
4. $k=k+1,\alpha_k=\alpha,T_k=T$
5. 重复操作步骤 2-4 直至 T_k 仅包含根节点
6. 采用交叉验证法从树序列 $T_0,T_1,...,T_n$ 中选取最优子树 T_α

2 代码实现

考虑的数据集为贷款申请样本数据表，具体可见 excel 文档。

2.1 sklearn 实现

决策树的 sklearn 接口位于 sklearn.tree.DecisionTreeClassifier, 其文档可见**此处**。

注意： sklearn 中的接口使用的是 CART 的改进算法，同时也不支持属性变量 (scikit-learn uses an optimised version of the CART algorithm; however, scikit-learn implementation does not support categorical variables for now.)

sklearn 中的接口也不提供剪枝功能,提高泛化能力通过调节 max_depth,min_samples_split,min_samples_leaf 等参数实现。

准备数据：

```
import pandas as pd
data = pd.read_excel('data.xlsx')
X = data[['年龄', '有工作', '有自己的房子', '信贷情况']].values
Y = data[['类别']].values
data
```

```
##     年龄 有工作 有自己的房子 信贷情况  类别
## 0   青年   否       否   一般   0
## 1   青年   否       否    好   0
## 2   青年   是       否    好   1
## 3   青年   是       是   一般   1
## 4   青年   否       否   一般   0
## 5   中年   否       否   一般   0
## 6   中年   否       否    好   0
## 7   中年   是       是    好   1
## 8   中年   否       是  非常好   1
## 9   中年   否       是  非常好   1
## 10  老年   否       是  非常好   1
## 11  老年   否       是    好   1
## 12  老年   是       否    好   1
## 13  老年   是       否  非常好   1
## 14  老年   否       否   一般   0
```

同样，我们需要用 LabelEncoder 对属性变量编码。

```
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier

le_tran = [LabelEncoder() for _ in range(4)]
for i in range(4):
```

```
    le_tran[i].fit(X[:,i])
    X[:,i] = le_tran[i].transform(X[:,i])
```

```
clf = DecisionTreeClassifier()
clf.fit(X,Y)
```

其生成的决策树如下：

```
from sklearn.tree import export_text
tree_representation = export_text(clf,feature_names=list(data.columns[:-1]))
print(tree_representation)
```

```
## |--- 有自己的房子 <= 0.50
## |   |--- 有工作 <= 0.50
## |   |   |--- class: 0
## |   |--- 有工作 >  0.50
## |   |   |--- class: 1
## |--- 有自己的房子 >  0.50
## |   |--- class: 1
```

2.2 ID3 和 C4.5 决策树

决策树的节点定义如下，其主要存储如下信息：

- feature: 该节点将要对哪个特征进行分支
- value: 该节点的变量分布，如 { ：6, ：9}
- label: 该节点的标签
- entropy: 该节点的熵
- subnode: 以字典形式存储该节点的子节点

```
class DTNode:

    def __init__(self):
        self.feature = None
        self.value = {}
        self.label = None
        self.entropy = None
        self.subnode = {}
```

决策树可如下表示，类内主要的函数有：

- init 函数：初始化时，需给定 method(ID3 or C4.5) 和阈值 eps

- fit 函数：拟合决策树，其中使用了 build_tree 函数进行递归
- print_tree 函数：可视化决策树，其中使用 print_node 函数进行递归
- predict 函数：进行预测，其中使用 predict_sample 函数以可同时对多个样本同时进行预测
- purn 函数：对决策树剪枝，其中使用 cut_node 函数进行递归

```
import numpy as np
import math

class DecisionTree:

    def __init__(self,method,eps):
        self.root = None
        self.method = method
        self.eps = eps

    def compute_entropy(self,value):
        value = [v/sum(value) for v in value]
        value = [v for v in value if v!=0]
        value = [-v*math.log(v) for v in value]
        return sum(value)

    def fit(self,X,Y):
        self.var_num = X.shape[1]
        df = pd.concat([pd.DataFrame(X,columns = ['var'+str(i) for i in range(self.var_num)]),\
                  pd.DataFrame(Y,columns=['y'])],axis=1)
        self.root = DTNode()
        self.bulid_tree(self.root,df,method=self.method,eps=self.eps)


    def bulid_tree(self,node,df,method,eps=0):
        node.value = dict(df['y'].value_counts())

        ## 均属于一个类
        if len(node.value) == 1:
            node.label = list(node.value.keys())[0]
            node.entropy = 0
        ## 无特征
        elif df.shape[1] == 1:
            node.label = [key for key,value in node.value.items() \
```

```
                if value == max(node.value.values())][0]
            node.entropy = self.compute_entropy(list(node.value.values()))

        ## 需递归
        else:
            node.label = [key for key,value in node.value.items() \
                if value == max(node.value.values())][0]
            node.entropy = self.compute_entropy(list(node.value.values()))
            ### 计算特征的信息增益
            var_list = list(df.columns[:-1])
            info = {}
            if method == 'ID3':
                for var in var_list:
                    df_tmp = [l[1] for l in list(df[[var,'y']].groupby(var))]
                    cond_info = sum([self.compute_entropy(list(d['y'].value_counts()))*len(d)\
                         for d in df_tmp])/len(df)
                    info[var] = node.entropy-cond_info
            elif method == 'C4.5':
                for var in var_list:
                    df_tmp = [l[1] for l in list(df[[var,'y']].groupby(var))]
                    cond_info = sum([self.compute_entropy(list(d['y'].value_counts()))*len(d)\
                         for d in df_tmp])/len(df)
                    info[var] = (node.entropy-cond_info)/ \
                        self.compute_entropy(list(df[var].value_counts()))
            ### 选出特征
            max_info = max(list(info.values()))
            feature = [key for key in info.keys() if info[key]==max_info][0]

            if max_info>=eps:
                node.feature = feature

                ### 建立子树
                fea_space = list(np.unique(df[feature].values))
                new_col = [l for l in list(df.columns) if l!=feature]
                for fea in fea_space:
                    node.subnode[fea] = DTNode()
                    self.bulid_tree(node.subnode[fea],df.loc[df[node.feature]==fea,new_col],\
                                    method=method,eps=eps)
```

```
    def print_node(self,node,var_dict,layer=0):
        if len(node.subnode)==0:
            print('|'+'\t|'*layer+'---类别:'+str(node.label)+', 分布:'+str(node.value))
        else:
            for key,value in node.subnode.items():
                print('|'+'\t|'*layer+'---'+var_dict[node.feature]+':'+str(key))
                self.print_node(value,var_dict,layer+1)

    def print_tree(self,col_name=None):
        var_name = ['var'+str(i) for i in range(self.var_num)]
        if col_name is not None:
            var_dict = dict(zip(var_name,col_name))
        else:
            var_dict = dict(zip(var_name,var_dict))

        self.print_node(self.root,var_dict,layer=0)

    def predict_sample(self,new_X):
        node = self.root
        while len(node.subnode)!=0:
            try:
                node = node.subnode[new_X[int(node.feature[3])]]
            except:
                break
        return node.label

    def predict(self,new_X):
        return np.apply_along_axis(self.predict_sample,axis=1,arr=new_X)

    def cut_node(self,node,alpha):
        subnode_entropy = sum([sum(n.value.values())*n.entropy for n in node.subnode.values()])
        if subnode_entropy!=0:
            for subnode in node.subnode.values():
                if subnode.entropy!=0:
                    self.cut_node(subnode,alpha)

        subnode_entropy = sum([sum(n.value.values())*n.entropy for n in node.subnode.values()])
        if subnode_entropy!=0:
```

```
            delta = sum(node.value.values())*node.entropy-subnode_entropy
            if delta<=alpha*(len(node.subnode)-1):
                node.subnode={}
                node.feature = None

    def prun(self,alpha):
        self.cut_node(self.root,alpha)
```

对数据进行拟合和可视化结果如下：

```
clf = DecisionTree(method='ID3',eps=0)
clf.fit(X,Y)
clf.print_tree(col_name=list(data.columns[:-1]))
```

```
## |---有自己的房子:否
## |    |---有工作:否
## |    |   |---类别:0,分布:{0: 6}
## |    |---有工作:是
## |    |   |---类别:1,分布:{1: 3}
## |---有自己的房子:是
## |    |---类别:1,分布:{1: 6}
```

对新数据预测结果如下：

```
new_X = np.array([['中年', '是', '否', '好'],\
                  ['青年', '否', '否', '非常好']])
clf.predict(new_X)
```

```
## [1 0]
```

2.3 剪枝

对该决策树剪枝结果如下：

```
clf.prun(alpha=5)
clf.print_tree(col_name=list(data.columns[:-1]))
```

```
## |---类别:1,分布:{1: 9, 0: 6}
```

注意，该决策树不存在过拟合现象，因此需要给定一个较大的 α 值才会进行剪枝。且实践证明，当树的第二层被剪枝时，此时的超参数 α 也会对第一层的树进行剪枝。

2.4 CART 决策树

此处主要实现回归树，使用的数据集是汽车售价数据，数据如下：

```
import numpy as np
import pandas as pd
data = pd.read_csv('cars.csv')
data['type'] = data['type'].apply(lambda x:{'small':0,'midsize':1,'large':2}[x])
X = data[['type', 'price', 'mpg_city', 'passengers']].values
Y = data[['weight']].values
data.head()
```

```
##    type  price  mpg_city  passengers  weight
## 0     0   15.9        25           5    2705
## 1     1   33.9        18           5    3560
## 2     1   37.7        19           6    3405
## 3     1   30.0        22           4    3640
## 4     1   15.7        22           6    2880
```

CART 决策树的节点需要存储如下信息：

- cut_var：该节点的切分变量
- cut_point：该节点的切分点
- avg：该节点内样本的平均值
- depth：该节点的深度
- num：该节点的样本个数
- left,right：该节点的左子节点和右子节点

```
class CARTNode:

    def __init__(self):
        self.cut_var = None
        self.cut_point = None
        self.avg = None
        self.depth = None
        self.num = None
        self.left = None
        self.right = None
```

回归树内如下：

```
class RegressionTree:

    def __init__(self,max_depth=float('inf'),min_samples_leaf=1):
        self.max_depth = max_depth
        self.min_samples_leaf = min_samples_leaf

    def compute_loss(self,df_var,df_y,s):
        df = pd.concat([df_var,df_y],axis=1)
        df.columns = ['x','y']
        df_1 = df[df['x']<=s]
        df_2 = df[df['x']>s]
        c_1 = np.mean(df_1['y'])
        c_2 = np.mean(df_2['y'])
        loss = np.sum((df_1['y']-c_1).values**2)+np.sum((df_2['y']-c_2).values**2)
        return loss

    def fit(self,X,Y):
        self.var_num = X.shape[1]
        df = pd.concat([pd.DataFrame(X,columns = ['var'+str(i) for i in range(self.var_num)]),\
            pd.DataFrame(Y,columns = ['y'])],axis=1)
        self.root = CARTNode()
        self.build_tree(self.root,df)

    def build_tree(self,node,df,depth=0):
        node.avg = np.mean(df['y'])
        node.num = len(df)
        node.depth = depth

        if node.depth < self.max_depth and node.num > self.min_samples_leaf:

            ## 寻找切分变量和切分点
            cut = []
            for j in range(self.var_num):
                s_list = sorted(np.unique(df['var'+str(j)]))[:-1]
                for s in s_list:
                    loss = self.compute_loss(df[['var'+str(j)]],df[['y']],s)
                    cut.append([[j,s],loss])
```

```
            loss = [c[1] for c in cut]
            min_loss = min(loss)
            cut_var, cut_point = [c[0] for c in cut][loss.index(min_loss)]

            node.cut_var = 'var'+str(cut_var)
            node.cut_point = cut_point

            ## 递归
            node.left = CARTNode()
            self.build_tree(node.left,df[df[node.cut_var]<=node.cut_point],depth+1)
            node.right = CARTNode()
            self.build_tree(node.right,df[df[node.cut_var]>node.cut_point],depth+1)

    def print_node(self,node,var_dict,layer=0):
        if node.left is None and node.right is None:
            print('|'+'\t|'*layer+'---输出:'+str(round(node.avg,3))+', 样本个数:'+str(node.num))

        if node.left is not None:
            print('|'+'\t|'*layer+'---'+var_dict[node.cut_var]+'<='+str(node.cut_point))
            self.print_node(node.left,var_dict,layer+1)

        if node.right is not None:
            print('|'+'\t|'*layer+'---'+var_dict[node.cut_var]+'>'+str(node.cut_point))
            self.print_node(node.right,var_dict,layer+1)


    def print_tree(self,col_name=None):
        var_name = ['var'+str(i) for i in range(self.var_num)]
        if col_name is not None:
            var_dict = dict(zip(var_name,col_name))
        else:
            var_dict = dict(zip(var_name,var_dict))

        self.print_node(self.root,var_dict,layer=0)
```

假定高度最大为 4，叶节点内的样本个数最少为 5，回归树的结果如下：

```
clf = RegressionTree(max_depth=4,min_samples_leaf=5)
clf.fit(X,Y)
```

```
clf.print_tree(col_name=list(data.columns[:-1]))
```

```
## |---type<=0.0
## |    |---mpg_city<=29.0
## |    |   |---price<=9.2
## |    |   |   |---输出:2295.0,样本个数:4
## |    |   |---price>9.2
## |    |   |   |---mpg_city<=25.0
## |    |   |   |   |---输出:2603.0,样本个数:5
## |    |   |   |---mpg_city>25.0
## |    |   |   |   |---输出:2414.0,样本个数:5
## |    |---mpg_city>29.0
## |    |   |---price<=8.6
## |    |   |   |---输出:1887.5,样本个数:4
## |    |   |---price>8.6
## |    |   |   |---输出:2251.667,样本个数:3
## |---type>0.0
## |    |---mpg_city<=18.0
## |    |   |---price<=35.2
## |    |   |   |---price<=23.7
## |    |   |   |   |---输出:3988.333,样本个数:3
## |    |   |   |---price>23.7
## |    |   |   |   |---输出:3605.0,样本个数:6
## |    |   |---price>35.2
## |    |   |   |---输出:3996.667,样本个数:3
## |    |---mpg_city>18.0
## |    |   |---price<=18.2
## |    |   |   |---mpg_city<=19.0
## |    |   |   |   |---输出:3610.0,样本个数:1
## |    |   |   |---mpg_city>19.0
## |    |   |   |   |---输出:2993.333,样本个数:6
## |    |   |---price>18.2
## |    |   |   |---price<=26.7
## |    |   |   |   |---输出:3415.5,样本个数:10
## |    |   |   |---price>26.7
## |    |   |   |   |---输出:3535.0,样本个数:4
```