第十周学习笔记—决策树

2022-06-16

第一章　最大熵模型学习

1.1　引言

我们在投资的时常常讲不要把所有鸡蛋放在一个篮子里，这样可以降低风险，这其实就是最大熵原理的一个朴素说法，因为当我们遇到不确定性时，就要保留各种可能性。

在信息处理中，这个原理同样适用。

在数学上，这个原理被称为最大熵原理（the maximum entropy principle）。

同样的，当我们在回答，扔一个骰子的时候，每个面朝上的概率分别是多少？所有人都会回答说是 1/6。这个回答是正确的，但是为什么是 1/6 而不是其他呢？这里就应用到最大熵原理。对这个“一无所知”的骰子，假定每一面朝上的概率相同是最保险的做法。从信息论的角度来解释，就是保留了最大的不确定性，也就是让熵达到最大。

最大熵原理指出，当我们需要对一个随机事件的概率分布进行预测的时候，我们的预测应当满足已知的条件，而对未知的情况不做任何主观的假设。在这种情况下，概率分布最均匀，预测的风险最小。因为这时候概率分布的信息熵最大，所以人们称这种模型为“最大熵模型”。

1.2　预备知识

1.2.1　记号约定

本文中将大量用到概率统计中的记号，为统一起见，这里做一些约定：

(1) 用大写字母表示随机变量（如 X，对于一个六面的骰子，有 X∈1,2,3,4,5,6），用小写字母表示随机变量中某个具体的取值（如 X=x）。

(2) 用 P(X) 表示随机变量 X 的概率分布，用 P(X,Y) 表示随机变量 X,Y 的联合概率分布，用 P(Y|X) 表示已知随机变量 X 时随机变量 Y 的条件概率分布。

(3) 用 p(X=x) 表示 X 具体取某个取值的概率（如 p(X=1)=1/6），在不引起混淆的情况下，也将 p(X=x) 简记为 p(x)。

(4) 用 P(X=x,Y=y) 表示联合概率，p(Y=y|X=x) 表示条件概率，两者长分别简记为 p(x,y)、p(y|x)。

1.2.2　贝叶斯定理

贝叶斯定理是用来描述两个条件概率之间的关系。若记 P(A)、P(B) 分别表示事件 A 和事件 B 发生的概率，P(A|B) 表示事件 B 发生的情况下事件 A 发生的概率，P(A,B) 表示事件发生的概率，则有：

$$P(A|B) = \frac{P(A,B)}{P(B)}$$

$$P(B|A) = \frac{P(A,B)}{P(A)}$$

利用两个上式，进一步可得：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

这就是贝叶斯公式。

1.2.3　熵

熵（entropy）原来是热力学中的概念，后由香农引入到信息论中。在信息论和概率统计中，熵用来表示随机变量不确定性的度量。

定义 1.1 设 $X \in \{x_1, x_2, \cdots, x_n\}$ 为一个离散随机变量，其概率分布为 $p(X = x_i) = p_i$, $i = 1, 2, \cdots, n$, 则 X 的熵为

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

其中，当 $p_i = 0$ 时，定义 $0 \log 0 = 0$.

- **条件熵：** $\mathrm{H}(Y|X) = \sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p(x,y) \log \frac{p(x)}{p(x,y)}$
- **联合熵：** $\mathrm{H}(X,Y) = -\sum_{x} \sum_{y} P(x,y) \log_2 [P(x,y)]$

1.2.4　最大似然估计

若总体 X 属于离散性，其分布律 P{X=x}=p(x;θ),$\theta \in \Theta$ 的形式为已知，θ 为待估参数，Θ 是 θ 可能取值的范围。

设 $X_1, X_2, X_3, \cdots, X_n$ 是来自 X 的样本，则 $X_1, X_2, X_3, \cdots, X_n$ 的联合分布律为：

$$\Pi P(x;\theta)$$

又设 $x_1, x_2, x_3, \cdots, x_n$ 是相应于 $X_1, X_2, X_3, \cdots, X_n$ 的一个样本值，则 $X_1, X_2, X_3, \cdots, X_n$ 取到观察值 $x_1, x_2, x_3, \cdots, x_n$ 的概率，即事件：$\{X_1 = x_1, X_2 = x_2, X_3 = x_3, \cdots, X_n = x_n\}$ 发生的概率为：

$$L(\theta) = L\{x_1, x_2, \cdots, x_n; \theta\} = \Pi P(x_i;\theta), \theta \in \Theta$$

这一概率随着 θ 而发生变化，它是 θ 的函数，$L(\theta)$ 称为样本的似然函数（注意，这里 $x_1, x_2, x_3, \cdots, x_n$ 是一只的样本值，它们都是常数）

固定样本值 $x_1, x_2, x_3, \cdots, x_n$，在 θ 的可能取值范围内，找到使得似然函数 $L(\theta)$ 达到最大值的参数值 $\hat{\theta}$，作为参数 θ 的估计值。即取 $\hat{\theta}$ 使：

$$L(x_1, x_2, \cdots, x_n; \hat{\theta}) = max_{\theta \in \Theta} L(x_1, x_2, \cdots, x_n; \theta)$$

这样得到的 $\hat{\theta}$ 与样本值 $x_1, x_2, x_3, \cdots, x_n$ 有关，常记为 $\hat{\theta}(x_1, x_2, \cdots, x_n)$，称为参数 θ 的最大似然估计值，而相应的统计量 $\hat{\theta}(X_1, X_2, \cdots, X_n)$ 称为参数 θ 的最大似然估计量。

1.2.5　拉格朗日乘数

最大熵模型的数学推导中需用到拉格朗日乘子法，这里做一个简单介绍。

问题：求函数 z=f(X) 在条件:$\phi_i(x) = 0 (i = 1, 2, \cdots, m)$ 下的可能极值点，其中 $X = (x_1, x_2, \cdots, x_n) \in R^n$

利用拉格朗日乘子法，可将上述带约束的极值问题转换为无约束极值问题来进行求解，具体步骤如下：

（1）构造函数 $L(x) = f(x) + \sum_{i=1}^{m} \lambda_i \phi_i(x)$，其中 $\lambda_i (i = 1, 2, \cdots, m)$ 为拉格朗日乘子。

（2）求解方程组：

$$\frac{\partial L}{\partial x} = 0$$

$$\phi_i(x) = 0 (i = 1, 2, \cdots, m)$$

其中，$\frac{\partial L}{\partial x} = (\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \cdots, \frac{\partial L}{\partial x_n})^T$ 表示 L 关于 x 的梯度。

求解上述方程组可得 $x_1, x_2, \cdots, x_n; \lambda_1, \lambda_2, \cdots, \lambda_m$，其中 $(x_1, x_2, \cdots, x_n)$ 就是函数 f 可能的极值点。

1.3　最大熵原理

其主要思想是，在只掌握关于未知分布的部分知识时，应该选取符合这些知识但熵值最大的概率分布。因为在这种情况下，符合已知知识的概率分布可能不止一个。我们知道，熵定义的实际上是一个随机变量的不确定性，熵最大的时候，说明随机变量最不确定，换句话说，也就是随机变量最随机，对其行为做准确预测最困难。

从这个意义上讲，那么最大熵原理的实质就是，在已知部分知识的前提下，关于未知分布最合理的推断就是符合已知知识最不确定或最随机的推断，这是我们可以作出的唯一不偏不倚的选择，任何其它的选择都意味着我们增加了其它的约束和假设，这些约束和假设根据我们掌握的信息无法作出。

最大熵原理认为，学习概率模型时，在所有可能的概率模型分布中，熵最大的模型是最好的模型。通常用约束条件来确定概率模型的集合，而最大熵原理就是在满足约束条件的模型集合中选取熵最大的模型。

直观地，最大熵原理认为要选择的概率模型首先要满足已有的事实，即约束条件。在没有更多信息的情况下，那些不确定的部分都是“等可能的”。最大熵原理通过熵的最大化来表示等可能性。

还是举引言部分的色子的例子，我们知道在仍一枚色子的时，它每个面朝上的概率都是 1/6。但是假设这个色子被特殊处理过，已知 4 朝上的概率是 1/3。在这种情况下，每个面朝上的概率是多少？正确的答案是：

已知：

(1)$P(x=4)=\frac{1}{3}$

(2)$P(x=1)+P(x=2)+P(x=3)+P(x=4)+P(x=5)+P(x=6)=1$

则其他面朝上的概率为：$P(x=1)=P(x=2)=P(x=3)=P(x=5)=P(x=6)=[1-P(x=4)]/5$ =2/15。

也就是说，我们已知条件必须满足（即面朝上为 4 的概率是 1/3），而对于其他面朝上的概率无法知道，这时对于未知概率分布在猜测分布的时候不能带有任何主观的假设，于是认为其他面朝上的概率是均等的是才是客观正确的，这是因为它刚好符合最大熵原理。

最大熵原理思想简单朴素，导出的数学模型非常漂亮，具有很多良好的特性，因而其应用也十分广泛，如天体物理学、医药、自然语言处理（NLP）等领域。

1.4　最大熵模型

最大熵原理是统计学习的一般原理，将它应用到分类得到最大熵模型。

假设分类模型是一个条件概率分布 $P(Y|X)$，$X \in \chi \in R^n$ 代表输入，$Y \in \gamma$ 表示输出，χ 和 γ 分别表示输入和输出的集合。这个模型表示的是对于给定的输入变量 X，以条件概率 P(Y|X) 输出 Y。

下面给定一个训练数据集：

$$T = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\}$$

学习目标是用最大熵原理选择最好的分类模型。

1.4.1　特征函数

用特征函数（feature function）f(x,y) 描述输入 x 和输出 y 之间的某一个事实。其定义为：

$$f(x,y) = \begin{cases} 1, & \text{x与y满足某一事实} \\ 0, & \text{否则} \end{cases}$$

它是一个二值函数，当 x 和 y 满足这个事实时取值为 1，否则取值为 0.

这里，我们假定训练数据集有 n 个特征函数，即：

$$f_i(x,y), i = 1, 2, \cdots, n$$

例如：假设我们需要判断“打”字是动词还是量词，已知的训练集如下：

(x_1, y_1) = (一打火柴，量词)
(x_2, y_2) = (三打啤酒，量词)
(x_3, y_3) = (五打塑料袋，量词)
(x_4, y_4) = (打电话，动词)
(x_5, y_5) = (打篮球，动词)

通过观察我们发现，“打”前面为数字时，“打”是量词，“打”后面为名词时，“打”是动词。于是，我们就从训练数据集中提取两个特征，分别用特征函数表示为：

$$f_1(x,y) = \begin{cases} 1, & 若“打”前面为数字; \\ 0, & 否则. \end{cases}$$

$$f_2(x,y) = \begin{cases} 1, & 若“打”后面为名词; \\ 0, & 否则. \end{cases}$$

定了了这两个特征函数后，对于训练数据，我们就有以下事实：

$$f_1(x_1,y_1) = f_1(x_2,y_2) = f_1(x_3,y_3) = 1; f_1(x_4,y_4) = f_1(x_5,y_5) = 0; \cdots$$
$$f_2(x_1,y_1) = f_2(x_2,y_2) = f_2(x_3,y_3) = 0; f_2(x_4,y_4) = f_2(x_5,y_5) = 1; \cdots$$

1.4.2　约束条件

（1）经验分布

经验分布式指在训练集 T 进行统计得到的分布，用 $\hat{P}$ 表示。

这里，对于给定的训练数据集，我们可以确定联合分布 p(x,y) 的经验分布和边缘分布 p(x) 的经验分布，分别以 $\hat{P}(x,y)$ 和 $\hat{P}(x)$ 表示，其定义为：

$$\tilde{p}(x,y) = \frac{count(x,y)}{N}, \quad \tilde{p}(x) = \frac{count(x)}{N}$$

其中，count(x,y) 表示训练数据集中 (x,y) 出现的次数，count(x) 表示训练数据中输入 x 的次数，N 表示训练样本集的容量。

备注：这里 p(x,y) 和 p(x) 是未知的，而 $\hat{P}(x,y)$ 和 $\hat{P}(x)$ 是已知的常量。后面在数学公式推导过程中会经常出现，这里不要搞混淆了。

特征函数 f(x,y) 关于经验分布 $\hat{P}(x,y)$ 的期望值，用 $E_{\hat{P}}(f)$ 表示：

$$E_{\hat{P}}(f) = \sum_{x,y} \hat{P}(x,y) f(x,y)$$

特征函数 f(x,y) 关于模型 p(x,y) 的期望值，用 $E_p(f)$ 表示：

$$E_P(f) = \sum_{x,y} P(x,y) f(x,y)$$

这里，p(x,y) 是未知的，我们建模的目标是希望生成 p(y|x)，所以我们希望用 p(y|x) 来表示 p(x,y)。根据贝叶斯定理，我们有 p(x,y)=p(x)p(y|x)，而 p(x) 也是未知，这时，我们可以通过 $\hat{P}(x)$ 取一个近似值，带入上一个表达式，最终得到 $E_{\hat{P}}(f)$ 定义如下：

$$\mathrm{E}_p(f)=\sum_{x,y}\tilde{p}(x)p(y|x)f(x,y)$$

如果模型能够获取训练数据中的信息，那么就可以假设这两个期望值相等，即：

$$E_{\hat{P}}(f)=E_P(f)$$

或：

$$\sum_{x,y}\tilde{p}(x,y)f(x,y)=\sum_{x,y}\tilde{p}(x)p(y|x)f(x,y)$$

我们将上述公式作为模型学习的约束条件。假如有 n 个特征函数 $f_i(x,y),i=1,2,\cdots,n$ 那么就有 n 个约束条件。

1.4.3　最大熵模型定义

假设满足所有的约束条件的模型集合为：

$$\mathrm{C}\equiv\{p\in P|\mathrm{E}_p(f_i)=\mathrm{E}_{\check{p}}(f_i),\mathrm{i}=1,2,..,\mathrm{n}\}$$

定义在条件概率分布 p(y|x) 的条件熵为：

$$H(P)=-\sum_{x,y}\hat{P}(x)P(y|x)logP(y|x)$$

则模型集合 C 中条件熵 H(p) 最大的模型称为最大熵模型，式中的对数为自然对数。

1.5　最大熵模型求解

最大熵模型的学习过程就是求解最大熵模型的过程。我们定义了最大熵模型，最大熵模型可以形式化为约束最优化的问题。

对于给定的训练数据集 $T=\{(x_1,y_1),(x_2,y_2),\cdots,(x_N,y_N)\}$ 以及特征函数 $f_i(x,y),i=1,2,\cdots,n$ 最大熵模型的学习等价于约束最优化问题：

$$max_{p\in C}H(P)=-\sum_{x,y}\hat{P}(x)P(y|x)logP(y|x)$$

$$s.t\quad E_P(f_i)=E_{\hat{P}}(f_i),i=1,2,\cdots,n$$

$$\sum_y P(y|x) = 1$$

也可以表示为求解最小值问题：

$$min_{P \in C} - H(P) = \sum_{x,y} \hat{P}(x) P(y|x) log P(y|x)$$

$$s.t \quad E_P(f_i) = E_{\hat{P}}(f_i), i = 1, 2, \cdots, n$$

$$\sum_y P(y|x) = 1$$

求解上述两个最优化问题的解就可以得到最大熵模型。下面我们通过拉格朗日乘子将约束最优化问题转化为无约束最优化问题。

将原始问题转换成对偶问题，通过求解对偶问题得到原始问题的解。之所以转换为对偶问题求解是因为对偶问题往往更易于求解。

1.5.1　原始问题和对偶问题

引入拉格朗日乘子 $\lambda_0, \lambda_1, \cdots, \lambda_n$，定义拉格朗日函数为：$L(P, \lambda)$

$$L(P, \lambda) = -H(P) + \lambda_0 (1 - \sum_y P(y|x)) + \sum_{i=1}^{n} \lambda_i (T_i - E_P(f_i))$$

$$= \sum_{x,y} \hat{P}(x) P(y|x) log P(y|x) + \lambda_0 (1 - \sum_y P(y|x)) + \sum_{i=1}^{n} \lambda_i (T_i - \sum_{x,y} \hat{P}(x) P(y|x) f_i(x, y))$$

则原始问题为：

$$min_{P \in C} max_{\lambda} (P, \lambda$$

对偶问题是：

$$max_{\lambda} min P \in C(P, \lambda)$$

则 H(P) 是关于 P 的凸函数，则原始问题和对偶问题是等价的。所以要求解最大熵模型，只需要求解对偶问题即可。

首先，我们求解对偶问题内部的极小化问题 $min_{P \in C}(P, \lambda)$, 这个问题是关于 λ 的函数，将其记为：

$$\Phi(\lambda) = min_{P \in C}(P, \lambda) = L(P_{\lambda}, \lambda)$$

$\Phi(\lambda)$ 称为对偶问题。同时，将其解记为：

$$P_{\lambda} = argmin_{P \in C}(P, \lambda) = P_{\lambda}(y|x)$$

要求极值问题，我们可以通过偏导来求解。

具体地，求 $L(P, \lambda)$ 对 $P(y|x)$ 的偏导数。

$$\frac{\partial L(p,\lambda)}{\partial p(y|x)} = \sum_{x,y}\tilde{p}(x)(\log p(y|x)+1) - \sum_{y}\lambda_0 - \sum_{i=1}^{n}\left(\lambda_i \sum_{x,y}\tilde{p}(x)f_i(x,y)\right)$$

$$= \sum_{x,y}\tilde{p}(x)(\log p(y|x)+1) - \sum_{x}\tilde{p}(x)\sum_{y}\lambda_0 - \sum_{x,y}\left(\tilde{p}(x)\sum_{i=1}^{n}\lambda_i f_i(x,y)\right) \xrightarrow{\text{利用}} \sum_{x}\tilde{p}(x)=1$$

$$= \sum_{x,y}\tilde{p}(x)(\log p(y|x)+1) - \sum_{x,y}\tilde{p}(x)\lambda_0 - \sum_{x,y}\left(\tilde{p}(x)\sum_{i=1}^{n}\lambda_i f_i(x,y)\right) \xrightarrow{\text{提取}} \tilde{p}(x)$$

$$= \sum_{x,y}\tilde{p}(x)\left(\log p(y|x)+1-\lambda_0 - \sum_{i=1}^{n}\lambda_i f_i(x,y)\right)$$

令偏导数为 0，即：

$$logP(y|x)+1-\lambda_0-\sum_{i=1}^{n}\lambda_i f_i(x,y)=0$$

在 $\hat{P}(x)>0$ 情况下，得到：

$$P(y|x)=e^{\sum_{i=1}^{n}\lambda_i f_i(x,y)+\lambda_0-1}=\frac{e^{\sum_{i=1}^{n}\lambda_i f_i(x,y)}}{e^{(1-\lambda_0}}$$

又知道约束条件 $\sum_y P(y|x)=1$，代入上式得：

$$\sum_y P(y|x)=\frac{1}{e^{1-\lambda_0}}\sum_y e^{\sum_{i=1}^{n}\lambda_i f_i(x,y)}=1$$

即：

$$e^{1-\lambda_0}=\sum_y e^{\sum_{i=1}^{n}\lambda_i f_i(x,y)}$$

带回上式得：

$$P_\lambda(y|x)=\frac{e^{\sum_{i=1}^{n}\lambda_i f_i(x,y)}}{Z_\lambda(x)}$$

其中，$Z_\lambda(x)=\sum_y e^{\sum_{i=1}^{n}\lambda_i f_i(x,y)}$

其中 $Z_\lambda(x)$ 称为规范化因子，$f_i(x,y)$ 是特征函数，λ_i 是特征的权值。$P_\lambda=P_\lambda(y|x)$ 就是最大熵模型，这里 λ 是最大熵模型中的参数向量。

接下来，就是求解对偶问题外部的极大化问题：

$$max_\lambda \Phi(\lambda)$$

将其解写为：

$$\lambda^*=argmax_\lambda\Phi(\lambda)$$

所以最大熵模型的解为（学习到的最优化模型）：

$$P^*=P\lambda^*=P\lambda^*(y|x)$$

所以，最大熵模型的学习归结为对偶问题 $\Phi(\lambda)$ 的极大化。

1.5.2　极大似然估计

下面我们证明对偶函数的极大化等价于最大熵模型的极大似然估计。

在上面，我们得到下式：

$$p_\lambda(y|\mathrm{x}) = \frac{\exp(\sum_{i=1}^{n} \lambda_i f_i(x,y))}{Z_\lambda(x)}$$

$$\text{其中} Z_\lambda(x) = \sum_y \exp\left(\sum_{i=1}^{n} \lambda_i f_i(x,y)\right)$$

将其做一个变换，得到如下公式：

$$\log p_\lambda(y|\mathrm{x}) = \sum_{i=1}^{n} \lambda_i f_i(x,y) - Z_\lambda(x)$$

将上式代入 $\Phi(\lambda) = min_{P\in C}(P,\lambda) = L(P_\lambda,\lambda)$，可得：

$$\begin{aligned}
&L(p_\lambda,\lambda) = \sum_{x,y} \tilde{p}(x)p_\lambda(y|x)\log p_\lambda(y|x) + \sum_{i=1}^{n} \lambda_i \left(T_i - \sum_{x,y} \tilde{p}(x)p_\lambda(y|x)f_i(x,y)\right) \\
&= \sum_{i=1}^{n} \lambda_i T_i + \sum_{x,y} \tilde{p}(x)p_\lambda(y|x)\left(\log p_\lambda(y|x) - \sum_{x,y} \lambda_i f_i(x,y)\right) \\
&= \sum_{i=1}^{n} \lambda_i T_i - \sum_{x,y} \tilde{p}(x)p_\lambda(y|x)\log Z_\lambda(x) \\
&= \sum_{i=1}^{n} \lambda_i T_i - \sum_{y} p_\lambda(y|x) \sum_{x} \tilde{p}(x)\log Z_\lambda(x) \\
&= \sum_{i=1}^{n} \lambda_i T_i - \sum_{x} \tilde{p}(x)\log Z_\lambda(x)
\end{aligned}$$

已知训练数据的经验概率分布为 $\hat{P}(x,y)$，条件概率 $P(y|x)$ 的对数似然函数为：

$$L_{\hat{P}}(P) = log\Pi_{x,y}P(y|x)^{\hat{P}(x,y)} = \sum_{x,y} logP(y|x)$$

推导后得到：

$$
\begin{aligned}
L_{\tilde{p}}(p) &= \sum_{x,y} \tilde{p}(x,y) \log p(y|x) \\
&= \sum_{x,y} \tilde{p}(x,y) \left(\sum_{i=1}^{n} \lambda_i f_i(x,y) - \log Z_\lambda(x) \right) \\
&= \sum_{x,y} \tilde{p}(x,y) \sum_{i=1}^{n} \lambda_i f_i(x,y) - \sum_{x,y} \tilde{p}(x,y) \log Z_\lambda(x) \\
&= \sum_{i=1}^{n} \lambda_i \sum_{x,y} \tilde{p}(x,y) f_i(x,y) - \sum_{x,y} \tilde{p}(x,y) \log Z_\lambda(x) \\
&= \sum_{i=1}^{n} \lambda_i T_i - \sum_{x,y} \tilde{p}(x,y) \log Z_\lambda(x) \xrightarrow{\text{利用}} \sum_{x,y} \tilde{p}(x,y) = \sum_{x} \tilde{p}(x) \\
&= \sum_{i=1}^{n} \lambda_i T_i - \sum_{x} \tilde{p}(x) \log Z_\lambda(x)
\end{aligned}
$$

所以，这说明对偶函数 $\Phi(\lambda)$ 的极大化和最大似然估计是等价的。

这样，最大熵学习问题就转换为具体求解对数似然函数极大化或对偶函数极大化的问题。

总结：模型学习就是在给定的训练数据条件下对模型进行极大似然估计或正则化的极大似然估计。

1.6　最大熵模型学习算法

1.6.1　通用迭代尺度法（Generalized Iterative Scaling, GIS）

已知最大熵模型为：

$$
p_\lambda(y|x) = \frac{\exp\left(\sum_{i=1}^{n} \lambda_i f_i(x,y)\right)}{Z_\lambda(x)}
$$

$$
\text{其中} Z_\lambda(x) = \sum_{y} \exp\left(\sum_{i=1}^{n} \lambda_i f_i(x,y) \right)
$$

GIS 算法流程如下：

算法 (GIS)

Step 1　初始化参数.$\lambda := 0$

Step 2　初始化参数.$\mathrm{E}_{\tilde{p}}(f_i), i = 1,2,..,n$

Step 3　执行一次迭代，对参数做一次刷新.
计算$\mathrm{E}_p(f_i), i = 1,2,..,n$

$For\ i = 1,2,..,n\ \ DO$
{
$\lambda_i := \lambda_i + \eta \log \frac{\mathrm{E}_{\tilde{p}}(f_i)}{\mathrm{E}_p(f_i)}$
}

Step 4　检查收敛条件，若达到收敛条件则算法结束；否则转至Step3

备注：收敛的条件可以是判断 λ_i 前后两次的差值是否足够小。

其他 η 类似于学习率，在实际应用中一般取 1/C。其中 C 一般取所有样本数据中最大的特征数量，即 $C = max_{x,y} \sum_{i=1}^{n} f_i(x,y)$

其中：

$$\mathrm{E}_{\tilde{p}}(f) = \sum_{x,y} \tilde{p}(x,y) f(x,y) = \frac{1}{N} \sum_{j=1}^{N} f_i(x_j, y_j)$$

$$\mathrm{E}_p(f) = \sum_{x,y} \tilde{p}(x) p(y|x) f(x,y) \cong \frac{1}{N} \sum_{j=1}^{N} \sum_{y} p^{(n)}(y|x_j) f_i(x_j, y_j)$$

其中，$p^{(n)}(y|x) \frac{\exp(\sum_{i=1}^{n} \lambda_i f_i(x,y))}{Z(x)}$

其大致原理可以概括为以下几个步骤：

（1）假定第 0 次迭代的初始模型为等概率的均匀分布。

（2）用第 N 次迭代的模型来估算每种信息特征在训练数据中的分布，如果超过了实际的，就把相应的模型参数变小；否则将它们变大。

（3）重复步骤（2）直至收敛。

1.7　最大熵模型小结

定义条件熵　$$H(y|x)=-\sum_{(x,y)\in z} p(y,x)\log p(y|x)$$

模型目的　$$p^*(y|x)=\arg\max_{p(y|x)\in P} H(y|x)$$

定义特征函数　$$f_i(x,y)\in\{0,1\}\quad i=1,2,\cdots,m$$

约束条件　$$\sum_{y\in Y} p(y|x)=1 \tag{1}$$

$$E(f_i)=\tilde{E}(f_i)\quad i=1,2,\cdots,m \tag{2}$$

$$\tilde{E}(f_i)=\sum_{(x,y)\in z}\tilde{p}(x,y)f_i(x,y)=\frac{1}{N}\sum_{(x,y)\in T} f_i(x,y)\qquad N=|T|$$

$$E(f_i)=\sum_{(x,y)\in z} p(x,y)f_i(x,y)=\sum_{(x,y)\in z} p(x)p(y|x)f_i(x,y)$$

最大熵模型的优点有：

(1) 最大熵统计模型获得的是所有满足约束条件的模型中信息熵极大的模型，作为经典的分类模型时准确率较高。

(2) 可以灵活地设置约束条件，通过约束条件的多少可以调节模型对未知数据的适应度和对已知数据的拟合程度。

最大熵模型的缺点有：

由于约束函数数量和样本数目有关系，导致迭代过程计算量巨大，实际应用比较难。

第二章　常见决策树算法 (ID3、C4.5、CART)

决策树学习采用的是自顶向下的递归方法，其基本思想是以信息熵为度量构造一颗熵值下降最快的树，到叶子节点处，熵值为 0。其具有可读性、分类速度快的优点，是一种有监督学习。

2.1　决策树（ID3、C4.5 和 CART 算法）

2.1.1　决策树是什么

决策树呈树形结构，在分类问题中，表示基于特征对实例进行分类的过程。学习时，利用训练数据，根据损失函数最小化的原则建立决策树模型；预测时，对新的数据，利用决策模型进行分类。

决策树的分类：决策树可以分为两类，主要取决于它目标变量的类型。

(1) 离散性决策树：离散性决策树，其目标变量是离散的，如性别：男或女等；

(2) 连续性决策树：连续性决策树，其目标变量是连续的，如工资、价格、年龄等；

决策树相关的重要概念：

（1）根结点 (Root Node)：它表示整个样本集合，并且该节点可以进一步划分成两个或多个子集。

（2）拆分 (Splitting)：表示将一个结点拆分成多个子集的过程。

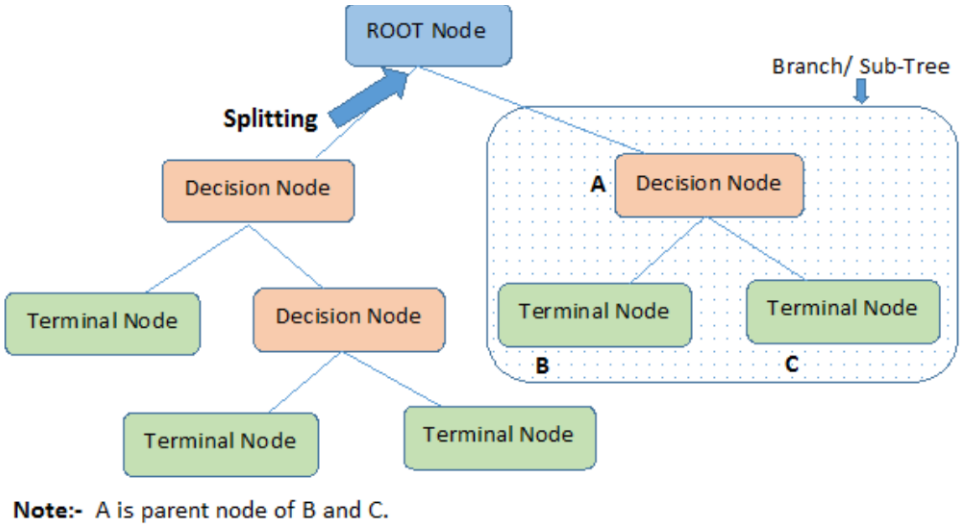
（3）决策结点 (Decision Node)：当一个子结点进一步被拆分成多个子节点时，这个子节点就叫做决策结点。

（4）叶子结点 (Leaf/Terminal Node)：无法再拆分的结点被称为叶子结点。

（5）剪枝 (Pruning)：移除决策树中子结点的过程就叫做剪枝，跟拆分过程相反。

（6）分支/子树 (Branch/Sub-Tree)：一棵决策树的一部分就叫做分支或子树。

（7）父结点和子结点 (Paren and Child Node)：一个结点被拆分成多个子节点，这个结点就叫做父节点；其拆分后的子结点也叫做子结点。



2.1.2　决策树的构造过程

决策树的构造过程一般分为 3 个部分，分别是特征选择、决策树生产和决策树裁剪。

（1）特征选择：

特征选择表示从众多的特征中选择一个特征作为当前节点分裂的标准，如何选择特征有不同的量化评估方法，从而衍生出不同的决策树，如 ID3（通过信息增益选择特征）、C4.5（通过信息增益比选择特征）、CART（通过 Gini 指数选择特征）等。

目的（准则）：使用某特征对数据集划分之后，各数据子集的纯度要比划分钱的数据集 D 的纯度高（也就是不确定性要比划分前数据集 D 的不确定性低）

（2）决策树的生成

根据选择的特征评估标准，从上至下递归地生成子节点，直到数据集不可分则停止决策树停止生长。这个过程实际上就是使用满足划分准则的特征不断的将数据集划分成纯度更高，不确定行更小的子集的过程。对于当前数据集的每一次划分，都希望根据某个特征划分之后的各个子集的纯度更高，不确定性更小。

（3）决策树的裁剪

决策树容易过拟合，一般需要剪枝来缩小树结构规模、缓解过拟合。

2.1.3　决策树的优缺点

决策树的优点：

（1）具有可读性，如果给定一个模型，那么过呢据所产生的决策树很容易推理出相应的逻辑表达。

（2）分类速度快，能在相对短的时间内能够对大型数据源做出可行且效果良好的结果。

决策树的缺点：

（1）对未知的测试数据未必有好的分类、泛化能力，即可能发生过拟合现象，此时可采用剪枝或随机森林。

2.1.4　ID3 算法原理

ID3 算法的核心是在决策树各个节点上应用信息增益准则选择特征递归地构建决策树。

信息增益

（1）熵

在信息论中，熵（entropy）是随机变量不确定性的度量，也就是熵越大，则随机变量的不确定性越大。设 X 是一个取有限个值得离散随机变量，其概率分布为：

$$P(X = x_i) = P_i, i = 1, 2, \cdots, n$$

则随机变量 X 的熵定义为：

$$H(X) = -\sum_{i=1}^{n} P_i log P_i$$

（2）条件熵

设有随机变量（X, Y），其联合概率分布为：

$$P(X = x_i, Y = y_i) = P_{ij}, i = 1, 2, \cdots, n, j = 1, 2, \cdots, m$$

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。随机变量 X 给定的条件下随机变量 Y 的条件熵 $H(Y|X)$，定义为 X 给定条件下 Y 的条件概率分布的熵对 X 的数学期望：

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

其中 $P_i = P(X = x_i), i = 1, 2, \cdots, n$

当熵和条件熵中的概率由数据估计得到时（如极大似然估计），所对应的熵与条件熵分别称为经验熵和经验条件熵。

（3）信息增益

定义：信息增益表示由于得知特征 A 的信息后儿时的数据集 D 的分类不确定性减少的程度，定义为：

$$Gain(D, A) = H(D) - H(D|A)$$

即集合 D 的经验熵 H(D) 与特征 A 给定条件下 D 的经验条件熵 H(H|A) 之差。

理解：选择划分后信息增益大的作为划分特征，说明使用该特征后划分得到的子集纯度越高，即不确定性越小。因此我们总是选择当前使得信息增益最大的特征来划分数据集。

缺点：信息增益偏向取值较多的特征（原因：当特征的取值较多时，根据此特征划分更容易得到纯度更高的子集，因此划分后的熵更低，即不确定性更低，因此信息增益更大）

ID3 算法

输入：训练数据集 D，特征集 A，阈值 ϵ；

输出：决策树 T

Step1: 若 D 中所有示例属于同一类 C_k，则 T 为单结点树，并将类 C_k 作为该节点的类标记，返回 T；

Step2: 若 $A = \phi$，则 T 为单结点树，并将 D 中实例数最大的类 C_k 作为该节点的类标记，返回 T；

Step3：否则，2.1.1（3）计算 A 中个特征对 D 的信息增益，选择信息增益最大的特征 A_k

Step4：如果 A_g 的信息增益小于阈值 ϵ 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，返回 T

Step5：否则，对 A_g 的每一种可能值 a_i，依 $A_g = a_i$ 将 D 分割为若干非空子集 D_i，将 D_i 中实例数最大的类作为标记，构建子结点，由结点及其子树构成树 T，返回 T；

Step6：对第 i 个子节点，以 D_i 为训练集，以 $A - \{A_g\}$ 为特征集合，递归调用 Step1 step5，得到子树 T_i，返回 T_i

2.1.5　C4.5 算法原理

C4.5 算法与 ID3 算法很相似，C4.5 算法是对 ID3 算法做了改进，在生成决策树过程中采用信息增益比来选择特征。

信息增益比

我们知道信息增益会偏向取值较多的特征，使用信息增益比可以对这一问题进行校正。

定义：特征 A 对训练数据集 D 的信息增益比 GainRatio(D,A) 定义为其信息增益 Gain(D,A) 与训练数据集 D 的经验熵 H(D) 之比：

$$GainRatio(D,A) = \frac{Gain(D,A)}{H(D)}$$

C4.5 算法

C4.5 算法过程跟 ID3 算法一样，只是选择特征的方法由信息增益改成信息增益比。

2.1.6　CART 算法原理

Gini 指数

分类问题中，假设有 K 个类，样本点属于第 k 类的概率为 P_k，则概率分布的基尼指数定义为：

$$Gini(P) = \sum_{k=1}^{K} P_k(1-P_k) = 1 - \sum_{k=1}^{K} P_k^2$$

备注：P_k 表示选中的样本属于 k 类别的概率，则这个样本被分错的概率为 $(1-P_k)$。

对于给定的样本集合 D，其基尼指数为：

$$Gini(D) = 1 - \sum_{k=1}^{K} (\frac{|C_k|}{|D|})^2$$

备注：这里 C_k 是 D 中属于第 k 类的样本自己，K 是类的个数。

如果样本集合 D 根据特征 A 是否取某一可能值 a 被分割成 D1 和 D2 两部分，即：

$$D_1 = \{(x,y) \in D | A(x) = a\}, D_2 = D - D_1$$

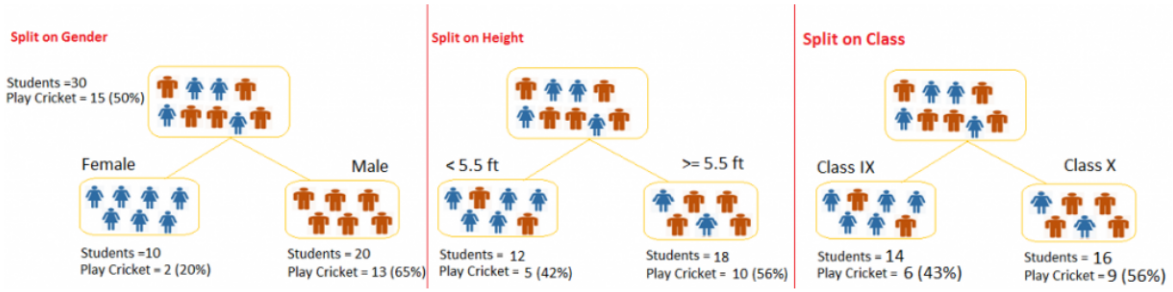
则在特征 A 的条件下，集合 D 的基尼指数定义为：

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

基尼指数 Gini(D) 表示集合 D 的不确定性，基尼指数 Gini(D,A) 表示经 A=a 分割后集合 D 的不确定性。基尼指数值越大，样本集合的不确定性也就越大，这一点跟熵相似。

下面举一个例子来说明上面的公式：

如下，是一个包含 30 个学生的样本，其包含三种特征，分别是：性别（男/女）、班级（IX/X）和高度（5 到 6ft）。其中 30 个学生里面有 15 个学生喜欢在闲暇时间玩板球。那么要如何选择第一个要划分的特征呢，我们通过上面的公式来进行计算。



如下，可以 Gini(D,Gender) 最小，所以选择性别作为最优特征。

$$\text{Gini}(D, Gender) = \frac{10}{30} * \left\{2 * \frac{2}{10} * \left(1 - \frac{2}{10}\right)\right\} + \frac{20}{30} * \left\{2 * \frac{13}{20} * \left(1 - \frac{13}{20}\right)\right\} = 0.41$$

$$\text{Gini}(D, Height) = \frac{12}{30} * \left\{2 * \frac{5}{12} * \left(1 - \frac{5}{12}\right)\right\} + \frac{18}{30} * \left\{2 * \frac{10}{18} * \left(1 - \frac{10}{18}\right)\right\} = 0.4907$$

$$\text{Gini}(D, Class) = \frac{14}{30} * \left\{2 * \frac{5}{14} * \left(1 - \frac{5}{14}\right)\right\} + \frac{16}{30} * \left\{2 * \frac{9}{16} * \left(1 - \frac{9}{16}\right)\right\} = 0.4768$$

CART 算法

输入：训练数据集 D，停止计算的条件

输出：CART 决策树

根据训练数据集，从根结点开始，递归地对每个结点进行以下操作，构建二叉树：

Step1：设结点的训练数据集为 D，计算现有特征对该数据集的基尼指数。此时，对每一个特征 A，对其可能取的每个值 a，根据样本点 A=a 的测试为“是”或“否”将 D 分割为 D1 和 D2 两部分，利用上式 Gini(D,A) 来计算 A=a 时的基尼指数。

Step2：在所有可能的特征 A 以及他们所有可能的切分点 a 中，选择基尼指数最小的特征及其对应可能的切分点作为最有特征与最优切分点。依最优特征与最有切分点，从现结点生成两个子节点，将训练数据集依特征分配到两个子节点中去。

Step3：对两个子结点递归地调用 Step1、Step2，直至满足条件。

Step4：生成 CART 决策树

算法停止计算的条件是节点中的样本个数小于预定阈值，或样本集的基尼指数小于预定阈值，或者没有更多特征。

第三章　决策树实战

本次学习的主要目的是为了结合李航老师的《统计学习方法》以及周志华老师的西瓜书的理论进行学习。因此，该笔记主要详细进行代码解析，从而透析在进行一项机器学习任务时候的思路，同时也积累自己的编程能力。

3.1　决策树的构造

优点：计算复杂度不高，输出结果易于理解，对中间值的缺失不敏感，可以处理不相关特征数据。

缺点：可能会产生过度匹配问题。

适用数据类型：数值型和标称型。

在构造决策树时，我们需要解决的第一个问题就是，当前数据集上哪个特征在划分数据分类时起决定性作用。为了找到决定性的特征，划分出最好的结果，我们必须评估每个特征。

完成测试之后，原始数据集就被划分为几个数据子集。这些数据子集会分布在第一个决策点的所有分支上。如果某个分支下的数据属于同一类型，则当前无需阅读的垃圾邮件已经正确地划分数据分类，无需进一步对数据集进行分割。如果数据子集内的数据不属于同一类型，则需要重复划分数据子集的过程。如何划分数据子集的算法和划分原始数据集的方法相同，直到所有具有相同类型的数据均在一个数据子集内。

创建分支的伪代码函数如下所示：

```
检测数据集中的每个子项是否属于同一分类：
    If so return 类标签；
    Else
        寻找划分数据集的最好特征
        划分数据集
        创建分支节点
            for 每个划分的子集
```

```
            调用函数并增加返回结果到分支节点中
    return 分支节点
```

决策树的一般流程:

(1) 收集数据：可以使用任何方法。

(2) 准备数据：树构造算法只适用于标称型数据，因此数值型数据必须离散化。

(3) 分析数据：可以使用任何方法，构造树完成之后，我们应该检查图形是否符合预期。

(4) 训练算法：构造树的数据结构。

(5) 测试算法：使用经验树计算错误率。

(6) 使用算法：此步骤可以适用于任何监督学习算法，而使用决策树可以更好地理解数据的内在含义。

3.2　信息增益

划分数据集的大原则是：将无序的数据变得更加有序。我们可以使用多种方法划分数据集，但是每种方法都有各自的优缺点。组织杂乱无章数据的一种方法就是使用信息论度量信息，信息论是量化处理信息的分支科学。我们可以在划分数据之前或之后使用信息论量化度量信息的内容。

在划分数据集之前之后信息发生的变化称为信息增益，知道如何计算信息增益，我们就可以计算每个特征值划分数据集获得的信息增益，获得信息增益最高的特征就是最好的选择。在可以评测哪种数据划分方式是最好的数据划分之前，我们必须学习如何计算信息增益。集合信息的度量方式称为香农熵或者简称为熵

熵定义为信息的期望值，在明晰这个概念之前，我们必须知道信息的定义。如果待分类的事务可能划分在多个分类之中，则符号 x_i 的信息定义为:

$$l(x_i) = -log_2 P(x_i)$$

其中 $P(x_i)$ 是选择该分类的概率。

为了计算熵，我们需要计算所有类别所有可能值包含的信息期望值，通过下面的公式得到：

$$H = -\sum_{i=1}^{n} P(x_i) log_2 P(x_i)$$

其中 n 是分类的数目。

3.2.1　计算给定数据集的信息熵

给定数据集为：

```
def create_dataset():
    dataset=[[1,1,'yes'],
             [1,1,'yes'],
             [1,0,'no'],
             [0,1,'no'],
             [0,1,'no']]
    labels=['no surfacing','flippers']
    return dataset,labels
```

该函数将书中海洋生物数据存在了一个 python 列表中，方便后续的处理。

```
my_dataset,labels=create_dataset()
print('=='*30)
print('my_dataset=',my_dataset)
print('labels=',labels)
```

我们的结果如下：

```
============================================================
my_dataset= [[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0,
                                        1, 'no']]
labels= ['no surfacing', 'flippers']
```

接下来我们定义一个 calculateentropy() 函数来计算香农信息熵：

```
#计算给定数据的香农熵
def calculate_entropy(dataset):
    num_entropy=len(dataset) #获取数据集样本个数
    num_labels={} #初始化一个字典用来保存每个标签出现的次数
    for feature_vector in dataset:
        current_label=feature_vector[-1] #逐个获取标签信息
        if current_label not in num_labels.keys(): # 如果标签没有放入统计次
                                                   数字典的话，就添加进去
            num_labels[current_label]=0
        num_labels[current_label]+=1
    entropy=0.0 #初始化香农熵
    for key in num_labels:
        prob=float(num_labels[key])/num_entropy #选择该标签的概率
        entropy-=prob*log(prob,2) #公式计算
```

```
    return entropy
```

首先，计算数据集中实例的总数。我们也可以在需要时再计算这个值，但是由于代码中多次用到这个值，为了提高代码效率，我们显式地声明一个变量保存实例总数。然后，创建一个数据字典，它的键值是最后一列的数值。如果当前键值不存在，则扩展字典并将当前键值加入字典。每个键值都记录了当前类别出现的次数。最后，使用所有类标签的发生频率计算类别出现的概率。我们将用这个概率计算香农熵，统计所有类标签发生的次数。下面我们看看如何使用熵划分数据集。

我们计算熵值：

```
entropy = calculate_entropy(my_dataset)
print('entropy=', entropy)
```

```
entropy= 0.9709505944546686
```

熵越高，则混合的数据也越多，我们可以在数据集中添加更多的分类，观察熵是如何变化的。这里我们增加第三个名为 maybe 的分类，测试熵的变化：

```
my_dataset[0][-1]='maybe'
print('my_dataset=',my_dataset)
print('entropy=',calculate_entropy(my_dataset))
```

```
entropy=1.3709505944546687
```

我们可以看到，在数据集中添加更多的分类，信息熵明显变大了。

3.3　划分数据集

我们将对每个特征划分数据集的结果计算一次信息熵，然后判断按照哪个特征划分数据集是最好的划分方式，下面我们先定义一个函数，用来实现按照给定的特征划分数据集这一功能：

```
def split_dataset(dataset,axis,value):
    new_dataset=[] #创建新列表以存放满足要求的样本
    for feature_vector in dataset:
        if feature_vector[axis]==value:
            #下面这两句用来将axis特征去掉，并将符合条件的添加到返回的数据集
                                                    中
            reduced_feature_vector=feature_vector[:axis]
```

```
            reduced_feature_vector.extend(feature_vector[axis+1:])
            new_dataset.append(reduced_feature_vector)
    return new_dataset
```

代码使用了三个输入参数：待划分的数据集、划分数据集的特征、需要返回的特征的值。Python 语言在函数中传递的是列表的引用，在函数内部对列表对象的修改，将会影响该列表对象的整个生存周期。为了消除这个不良影响，我们需要在函数的开始声明一个新列表对象。因为该函数代码在同一数据集上被调用多次，为了不修改原始数据集，创建一个新的列表对象。数据集这个列表中的各个元素也是列表，我们要遍历数据集中的每个元素，一旦发现符合要求的值，则将其添加到新创建的列表中。

在 if 语句中，程序将符合特征的数据抽取出来。后面讲述得更简单，这里我们可以这样理解这段代码：当我们按照某个特征划分数据集时，就需要将所有符合要求的元素抽取出来。

代码中使用了 Python 语言列表类型自带的 extend() 和 append() 方法。这两个方法功能类似，但是在处理多个列表时，这两个方法的处理结果是完全不同的。

```
In [3]: a=[1,2,3]
In [4]: b=[4,5,6]
In [5]: a.append(b)
In [6]: a
Out[6]: [1, 2, 3, [4, 5, 6]]
```

```
In [9]: a=[1,2,3]
In [10]: a.extend(b)
In [11]: a
Out[11]: [1, 2, 3, 4, 5, 6]
```

这段代码其实很简单，但是有个地方需要解释一下，就是：

```
reduced_feature_vector=feature_vector[:axis]
reduced_feature_vector.extend(feature_vector[axis+1:])
```

从上面这一波操作可以看出，通过第一步操作，可以将 axis 以前的元素存到 reduce-featurevector 列表中，而通过第二步操作，可以将 axis 以后的元素也同样存进去，这样就可以剔除 axis 了。

执行后得到划分后结果：

```
print('=='*30)
split_dataset = split_dataset(my_dataset, 0, 1)
```

```
#split_dataset = split_dataset(my_dataset, 0, 0)
print('split_dataset=', split_dataset)
print('=='*30)
```

```
============================================================
split_dataset= [[1, 'yes'], [1, 'yes'], [0, 'no']]
split_dataset= [[1, 'no'], [1, 'no']]
============================================================
```

我们可以很直观看出，通过最后两条命令，数据集通过“不浮出水面是否可以生存”这一特征被划分。

以上无论是用来计算香农信息熵的函数，还是用来划分数据集的函数，其实都是我们提前做好的两个“工具包”，因为我们从决策树的原理上理解也很容易看出，这两个函数的计算肯定不止一次，需要根据数据集的需要进行循环计算，并在前后评估信息增益，从而才能找到我们想要的结果——最优的数据集划分方法。

那么下面就进入了这一步，我们添加 choosebestfeaturetosplit 函数，我们先用 ID3 算法：

```
def choose_best_feature_to_split(dataset):
    number_features=len(dataset[0])-1 #获取样本集中特征个数，-1是因为最后一
                                        列是label
    base_entropy = calculate_entropy(dataset)  # 计算根节点的信息熵
    best_info_gain = 0.0  # 初始化信息增益
    best_feature = -1  # 初始化最优特征的索引值
    for i in range(number_features):  # 遍历所有特征，i表示第几个特征
        #将dataset中的数据按行依次放入example中，然后取得example中的example
                                                [i]元素，即获得特征i的所有取
                                                值
        feature_list = [example[i] for example in dataset]
        #由上一步得到了特征i的取值，比如[1,1,1,0,0]，使用集合这个数据类型删
                                                除多余重复的取值，则剩下[1,0]
        unique_vals = set(feature_list) # 获取无重复的属性特征值
        new_entropy=0.0
        for value in unique_vals:
            sub_dataset =split_dataset(dataset, i, value) #逐个划分数据集，
                                                得到基于特征i和对应的取值
                                                划分后的子集
            prob = len(sub_dataset)/float(len(dataset)) #根据特征i可能取值
                                                划分出来的子集的概率
```

```
            new_entropy += prob * calculate_entropy(sub_dataset) #求解分支
                                                    节点的信息熵
        info_gain = base_entropy - new_entropy  #计算信息增益，g(D,A)=H(D)-
                                                H(D|A)
        if (info_gain > best_info_gain):  # 对循环求得的信息增益进行大小比
                                          较
            best_info_gain=info_gain
            best_feature = i #如果计算所得信息增益最大，则求得最佳划分方法
    return best_feature #返回划分属性（特征）
```

知识要点：
①链表推导式：featList = [example[i] for example in dataSet]，高效简洁生成一个列表。
②set()：set() 函数创建一个无序不重复元素集，可进行关系测试，删除重复数据，还可以计算交集、差集、并集等。
③信息增益：《机器学习》（周志华 著）：假定离散属性a有V个可能取值，若使用a来对样本集D进行划分，则会产生V个分支节点，其中第v个分支节点包含了D中所有在属性a上取值为a^v的样本，记为D^v，则用属性a对样本集D进行划分所得到的信息增益定义为

$$Gain(D,a)=Ent(D)-\sum_{v=1}^{V}\frac{|D^v|}{|D|}Ent(D^v)$$

一般而言，信息增益越大，则意味着使用属性a来进行划分所得的“纯度提升”越大。其中$\frac{|D^v|}{|D|}$表示分支节点的权重，即在计算信息增益时，需要考虑分支的样本数占比，样本数越多的分支，影响越大，其对应的是函数chooseBestFeatureToSplit中的prob参数。

接下来我们对这个函数进行测试：

```
print('=='*30)
selection_1=choose_best_feature_to_split(my_dataset)
print('selection_1=',selection_1)
print('=='*30)
```

我们输出的结果如下：

```
============================================================
selection_1= 0
============================================================
```

代码运行后结果告诉我们，第 0 个特征是最好的用于划分数据集的特征。

3.4　递归构建决策树

我们已经按照例程分别针对计算信息熵、选择最佳划分属性各自创建了函数模块，似乎已经可以进行决策树的构建了，但是这里其实还有一个细节需要考虑，那就是当我们完成最后一个属性的划分时，很有可能会出现类标签不唯一的情况。而这种情况，书中给我们介绍了一种方式—多数表决。

下面我们就定义一个 majoritycnt 函数用来完成这一操作：

```
def majority_cnt(class_list):
    class_count={}
    for vote in class_list:
        if vote not in class_count.keys(): class_count[vote] = 0
        class_count[vote] += 1
    #分解为元组列表，operator.itemgetter(1)按照第二个元素的次序对元组进行排
                                          序，reverse=True是逆序，即按照从
                                          大到小的顺序排列
    sorted_class_count = sorted(class_count.iteritems(), key=operator.
                                          itemgetter(1), reverse=True)
    return sorted_class_count[0][0]
```

完成多数表决函数的创建，我们就可以开始构建一棵完整的决策树了：

```
def create_decision_tree(dataset,labels):
    class_list = [example[-1] for example in dataset] #获取类别标签
    if class_list.count(class_list[0]) == len(class_list):
        return class_list[0] #类别完全相同则停止继续划分
    if len(dataset[0]) == 1:
        return majority_cnt(class_list) #遍历完所有特征时返回出现次数最多的
                                          类别
    best_feature = choose_best_feature_to_split(dataset) #选取最优划分特征
    best_feature_label = labels[best_feature] #获取最优划分特征对应的属性标
                                          签
    my_tree = {best_feature_label:{}} #存储树的所有信息
    del(labels[best_feature]) #删除已经使用过的属性标签
    feature_values = [example[best_feature] for example in dataset] #得到训
                                          练集中所有最优特征的属性值
    unique_vals = set(feature_values) #去掉重复的属性值
    for value in unique_vals: #遍历特征，创建决策树
        sub_labels = labels[:] #剩余的属性标签列表
        my_tree[best_feature_label][value] = create_decision_tree(
```

```
                                                split_dataset(dataset,
                                                best_feature, value),
                                                sub_labels) #递归函数实现决策
                                                树的构建
    return my_tree
```

（属性标签1）no surfacing	（属性标签2）flippers	（类标签） Fish
1	1	yes
1	1	yes
1	0	no
0	1	no
0	1	no

这里的 labels=[“no surfacing”, “flippers”] 指的是属性标签，与类别标签是不同的。实际的决策树操作不需要用到我们的 labels，需要用到的是 createtree 函数第一行的 classlist 列表中所获取到的数据集最后一列参数，用来作为划分停止的条件以及当所有特征都被遍历完后输入 majoritycnt 多数表决函数获得最终的分类返回值。

知识要点：

(1)count()：Python count() 方法用于统计字符串里某个字符出现的次数。可选参数为在字符串搜索的开始与结束位置。

(2) 递归函数：在函数内部调用自己本身的函数。理论上，递归函数一般都可以写成循环的方式。下面这句代码就是 createtree 函数的核心：mytree[bestfeaturelabel][value] = createtree(splitdataset(dataset, bestfeature, value),sublabels)

另外要提一句的是，构建决策树的 myTree 是一个多层嵌套的字典，即字典内嵌套了多层字典，采用的是递归的方式来构建的。

我们现在来进行调试：

```
print('=='*30)
my_tree=create_decision_tree(my_dataset,labels)
print('my_tree=',my_tree)
print('=='*30)
```

```
============================================================
my_tree= {'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}
============================================================
```

从最终输出结果，我们可以清晰地看到，从左到右进行划分数据，总共包含了 3 个叶节点以及 2 个分支节点（判断节点）。

3.5　在 python 中使用 Matplotlib 注解绘制树形图

我们的代码如下：

```
"""
函数说明:绘制结点
Parameters:
    node_txt - 结点名
    center_pt - 文本位置
    parent_pt - 标注的箭头位置
    node_type - 结点格式
Returns:
    无
"""
def plot_node(node_txt, center_pt, parent_pt, node_type):
    arrow_args = dict(arrowstyle="<-") #定义箭头格式
    # 绘制结点
    create_plot.ax1.annotate(node_txt, xy=parent_pt, xycoords='axes
                                        fraction',xytext=center_pt,
                                        textcoords='axes fraction',va="
                                        center", ha="center", bbox=
                                        node_type, arrowprops=arrow_args)
"""
函数说明:获取决策树叶子结点的数目
Parameters:
    myTree - 决策树
Returns:
    numLeafs - 决策树的叶子结点的数目
"""
def get_num_leafs(my_tree):
    num_leafs = 0 #初始化叶子
    first_str = list(my_tree.keys())[0] ##python3中my——tree.keys()返回的
                                        是dict_keys,不在是list,所以不能使
                                        用myTree.keys()[0]的方法获取结点
                                        属性，可以使用list(myTree.keys())
                                        [0]
```

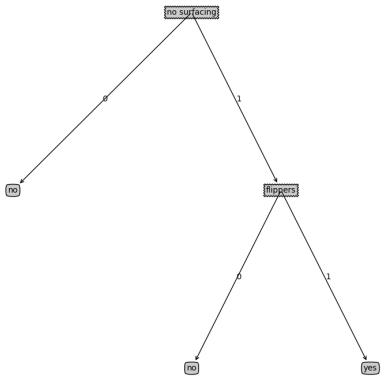
```
    second_dict = my_tree[first_str] ##获取下一组字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            num_leafs += get_num_leafs(second_dict[key])
        else:
            num_leafs += 1
    return num_leafs


"""
函数说明:获取决策树的层数
Parameters:
    myTree - 决策树
Returns:
    maxDepth - 决策树的层数
"""
def get_tree_depth(my_tree):
    max_depth = 0 #初始化决策树深度
    first_str = list(my_tree.keys())[0]
    second_dict = my_tree[first_str] #获取下一个字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            thisDepth = get_tree_depth(second_dict[key]) + 1
        else:
            thisDepth = 1
        if thisDepth > max_depth: #更新层数
            max_depth = thisDepth
    return max_depth


"""
函数说明:标注有向边属性值
Parameters:
    cntr_pt、parent_pt - 用于计算标注位置
    txt_string - 标注的内容
Returns:
    无
```

```
"""
def plot_mid_text(cntr_pt, parent_pt, txt_string):
    x_mid = (parent_pt[0] - cntr_pt[0]) / 2.0 + cntr_pt[0]
    y_mid = (parent_pt[1] - cntr_pt[1]) / 2.0 + cntr_pt[1]
    create_plot.ax1.text(x_mid, y_mid, txt_string)

"""
函数说明:绘制决策树
Parameters:
    my_tree - 决策树(字典)
    parent_pt - 标注的内容
    node_txt - 结点名
Returns:
    无
"""

leaf_node = dict(boxstyle="round4", fc="0.8")
decision_node = dict(boxstyle="sawtooth", fc="0.8")

def plot_tree(my_tree, parent_pt, node_txt):
    num_leafs = get_num_leafs(my_tree)
    depth = get_tree_depth(my_tree)
    first_str = list(my_tree.keys())[0]
    cntr_pt = (plot_tree.x_off + (1.0 + float(num_leafs)) / 2.0 / plot_tree
                                          .total_w, plot_tree.y_off)
    plot_mid_text(cntr_pt, parent_pt, node_txt)
    plot_node(first_str, cntr_pt, parent_pt, decision_node)
    second_dict = my_tree[first_str]
    plot_tree.y_off = plot_tree.y_off - 1.0 / plot_tree.total_d
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
            plot_tree(second_dict[key], cntr_pt, str(key))
        else:
            plot_tree.x_off = plot_tree.x_off + 1.0 / plot_tree.total_w
            plot_node(second_dict[key], (plot_tree.x_off, plot_tree.y_off),
                                                cntr_pt, leaf_node)
            plot_mid_text((plot_tree.x_off, plot_tree.y_off), cntr_pt, str(
                                               key))
    plot_tree.y_off = plot_tree.y_off + 1.0 / plot_tree.total_d
```

```
"""
函数说明:创建绘制面板
Parameters:
    in_tree - 决策树(字典)
Returns:
    无
"""
def create_plot(in_tree):
    fig = plt.figure(1,figsize=(12,8),facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    create_plot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plot_tree.total_w = float(get_num_leafs(in_tree))
    plot_tree.total_d = float(get_tree_depth(in_tree))
    plot_tree.x_off = -0.5 / plot_tree.total_w
    plot_tree.y_off = 1.0
    plot_tree(in_tree, (0.5, 1.0), '')
    plt.show()
```

不出意外的话，我们就可以得到如下结果，可以看到决策树绘制完成。plotnode 函数的工作就是绘制各个结点，包括内结点和叶子结点。plotmidtext 函数的工作就是绘制各个有向边的属性，例如各个有向边的 0 和 1。这部分内容呢，个人感觉可以选择性掌握，能掌握最好，不能掌握可以放一放。

我们绘制自己创建的数据集的可视化图：



3.6　使用决策树执行分类

我们已经实现了决策树算法，并通过可视化的方式了解了数据的真实含义，下面我们就来学习一下如何通过我们的算法来构建一个分类器，classify 函数需要 3 个输入参数，分别是 inputtree（我们通过决策树迭代学习所得到的 mytree）、featlabels（属性标签 labels）、testvec（输入的测试样本），具体实现方式如下（下面有坑，后面填）：

```
def classify(input_tree,feature_labels,test_vector):
    first_string = list(input_tree.keys())[0] #获取根节点
    second_dict = input_tree[first_string] #获取下一级分支
    feature_index = feature_labels.index(first_string) #查找当前列表中第一
                                                个匹配firstStr变量的元素的索引
    key = test_vector[feature_index] #获取测试样本中，与根节点特征对应的取
                                                值
    value_feature = second_dict[key]#获取测试样本通过第一个特征分类器后的输
                                                出
    if isinstance(value_feature, dict): #判断节点是否为字典来以此判断是否为
                                                叶节点
        class_label = classify(value_feature, feature_labels, test_vector)
    else:
        class_label = value_feature#如果到达叶子节点，则返回当前节点的分类
                                                标签
    return class_label
```

知识要点：

(1)index()：Python index() 方法检测字符串中是否包含子字符串 str。

(2)isinstance()：isinstance(object, classinfo) 是用来判断一个对象是否是一个已知的类型。其中参数有：

- object -实例对象
- classinfo -可以是直接或间接类名、基本类型或者由它们组成的元组

如果对象的类型与参数二的类型（classinfo）相同则返回 True，否则返回 False。

定义好分类器之后，我们测试一下：

```
my_dataset, labels = create_dataset()
result=classify(my_tree,labels,[1,0])
print('result=',result)
```

```
result= no
```

3.7　使用算法：决策树的存储

构造决策树是很耗时的，为了节省时间，我们可以在每次执行分类时直接调用已经构造好的决策树。为了解决这一问题，需要使用 python 模块 pickle 序列化对象。

知识要点：

序列化：把变量从内存中变成可存储或传输的过程称之为序列化，序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上了。反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化。

大致了解了序列化之后，我们就来定义 2 个函数进行决策树的序列化和反序列化：

```
"""
函数说明:存储决策树
Parameters:
    inputTree - 已经生成的决策树
    filename - 决策树的存储文件名
Returns:
    无
"""
def store_tree(input_tree,file_name):
    import pickle
    with open(file_name,'wb') as fw:
        pickle.dump(input_tree,fw)

"""
函数说明:读取决策树
Parameters:
    file_name - 决策树的存储文件名
Returns:
    pickle.load(fr) - 决策树字典
"""
def grab_tree(file_name):
    import pickle
    fr = open(file_name, 'rb')
    return pickle.load(fr)
```

知识要点：

(1)pickle.dump()：序列化对象，将对象 obj 保存到文件 file 中去。

(2)pickle.load()：反序列化对象，将文件中的数据解析为一个 python 对象。

接下来我们测试算法：

```
store_tree(my_tree,'decision_tree_practice.txt')
myTree = grab_tree('decision_tree_practice.txt')
print(myTree)
```

```
{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}
```

3.8　使用决策树预测隐形眼镜类型

前面我们学习了决策树从数据集的构成、分类器的构建、数据可视化以及序列化操作，下面我们来通过“使用决策树预测隐形眼镜类型”这一案例来学习决策树应用于解决实际问题的思路。

数据集信息：

特征（4 个）：age（年龄）、prescript（症状）、astigmatic（是否散光）、tearRate（眼泪数量）隐形眼镜类别（3 个）：硬材质 (hard)、软材质 (soft)、不适合佩戴隐形眼镜 (no lenses)

```
0          young\tmyope\tno\treduced\tno lenses
1                young\tmyope\tno\tnormal\tsoft
2         young\tmyope\tyes\treduced\tno lenses
3               young\tmyope\tyes\tnormal\thard
4          young\thyper\tno\treduced\tno lenses
5                young\thyper\tno\tnormal\tsoft
6         young\thyper\tyes\treduced\tno lenses
7               young\thyper\tyes\tnormal\thard
8            pre\tmyope\tno\treduced\tno lenses
9                  pre\tmyope\tno\tnormal\tsoft
10          pre\tmyope\tyes\treduced\tno lenses
```

实现的代码如下：

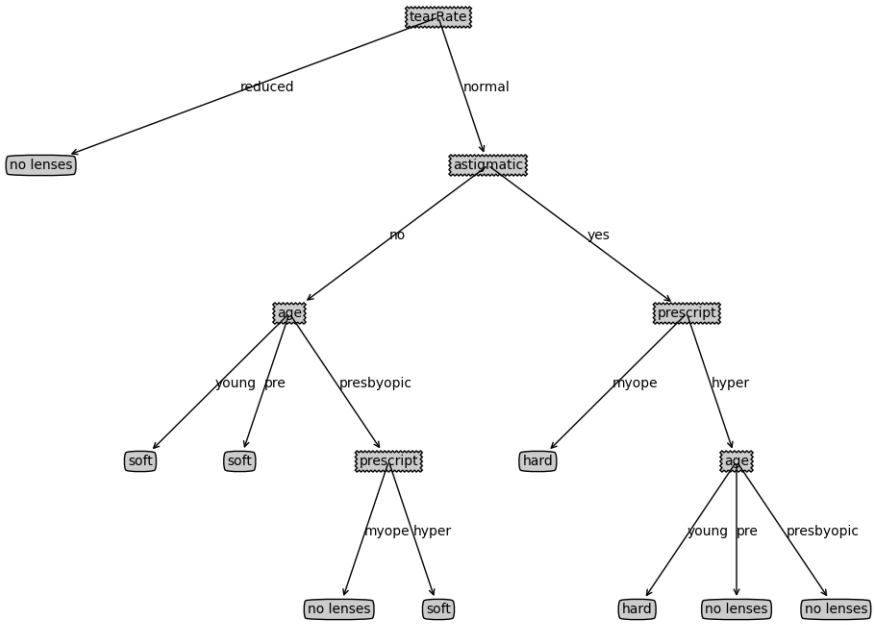
```
fr=open('lenses.txt')
lenses = [inst.strip().split('\t') for inst in fr.readlines()]
print('=='*30)
print('lenses=',lenses)
lensesLabels = ['age', 'prescript', 'astigmatic', 'tearRate']
lenses_tree = create_decision_tree(lenses, lensesLabels)
print('=='*30)
```

```
create_plot(lenses_tree)
```

结果展示如下：

```
============================================================
lenses= [['young', 'myope', 'no', 'reduced', 'no lenses'], ['young', 'myope
                                        ', 'no', 'normal', 'soft'], ['young',
                                         'myope', 'yes', 'reduced', 'no
                                        lenses'], ['young', 'myope', 'yes', '
                                        normal', 'hard'], ['young', 'hyper',
                                        'no', 'reduced', 'no lenses'], ['
                                        young', 'hyper', 'no', 'normal', '
                                        soft'], ['young', 'hyper', 'yes', '
                                        reduced', 'no lenses'], ['young', '
                                        hyper', 'yes', 'normal', 'hard'], ['
                                        pre', 'myope', 'no', 'reduced', 'no
                                        lenses'], ['pre', 'myope', 'no', '
                                        normal', 'soft'], ['pre', 'myope', '
                                        yes', 'reduced', 'no lenses'], ['pre'
                                        , 'myope', 'yes', 'normal', 'hard'],
                                        ['pre', 'hyper', 'no', 'reduced', 'no
                                         lenses'], ['pre', 'hyper', 'no', '
                                        normal', 'soft'], ['pre', 'hyper', '
                                        yes', 'reduced', 'no lenses'], ['pre'
                                        , 'hyper', 'yes', 'normal', 'no
                                        lenses'], ['presbyopic', 'myope', 'no
                                        ', 'reduced', 'no lenses'], ['
                                        presbyopic', 'myope', 'no', 'normal',
                                         'no lenses'], ['presbyopic', 'myope'
                                        , 'yes', 'reduced', 'no lenses'], ['
                                        presbyopic', 'myope', 'yes', 'normal'
                                        , 'hard'], ['presbyopic', 'hyper', '
                                        no', 'reduced', 'no lenses'], ['
                                        presbyopic', 'hyper', 'no', 'normal',
                                         'soft'], ['presbyopic', 'hyper', '
                                        yes', 'reduced', 'no lenses'], ['
                                        presbyopic', 'hyper', 'yes', 'normal'
                                        , 'no lenses']]
============================================================
```

决策树的可视化为：



我们可以看到，其实创建好分类器后，在面对实际问题的时候，最重要的一步无非就是将数据集处理成我们想要的格式，并与分类器的输入匹配。从上图我们可以看出，医生最多需要问四个问题就能够确认患者需要佩戴的隐形眼镜类型。

3.9　C4.5 算法

C4.5 算法跟 ID3 算法，不同的地方只是特征选择方法，即：

```
def choose_best_feature_to_split_2(dataset):
    """
    按照最大信息增益比划分数据
    :param dataset: 样本数据，如：  [[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, '
                                        no'], [0, 1, 'no'], [0, 1, 'no']]
    :return:
    """
    number_feature = len(dataset[0]) - 1 # 特征个数，如：不浮出水面是否可以
                                         生存 和是否有脚蹼
    base_entropy = calculate_entropy(dataset) # 经验熵H(D)
    best_info_gain = 0.0
    best_feature = -1
```

```
    for i in range(number_feature):
        feature_list = [number[i] for number in dataset]  # 得到某个特征下
                                                          所有值（某列）
        unique_feature_list = set(feature_list)  # 获取无重复的属性特征值
        new_entropy = 0
        split_info = 0.0
        for value in unique_feature_list:
            sub_dataset = split_dataset(dataset,i, value)
            prob = len(sub_dataset) / float(len(dataset))  # 即p(t)
            new_entropy += prob * calculate_entropy(sub_dataset)  # 对各子
                                                                  集香农熵求和
            split_info += -prob * log(prob, 2)
        info_gain = base_entropy - new_entropy  # 计算信息增益，g(D,A)=H(D)
                                                -H(D|A)
        if split_info == 0:  # fix the overflow bug
            continue
        info_gain = info_gain / split_info
        # 最大信息增益比
        if info_gain > best_info_gain:
            best_info_gain = info_gain
            best_feature = i
    return best_feature
```

效果跟 ID3 算法一样，这里就不重复。

这一章节我们主要学习的是决策树中的 ID3 算法，ID3 名字中的 ID 指的是 Iterative Dichotomiser（迭代二分器），这是一个很好的算法，但是它也存在很多问题，它是基于“信息增益最大化”来进行的，在许多场合下不免暴露其“贪心”本质。

第四章　源代码

```
from math import log
import operator
import matplotlib.pyplot as plt
import pandas as pd
#计算给定数据的香农熵
def calculate_entropy(dataset):
    num_entropy=len(dataset) #获取数据集样本个数
    num_labels={} #初始化一个字典用来保存每个标签出现的次数
    for feature_vector in dataset:
        current_label=feature_vector[-1] #逐个获取标签信息
        if current_label not in num_labels.keys(): # 如果标签没有放入统计次
                                                   数字典的话，就添加进去
            num_labels[current_label]=0
        num_labels[current_label]+=1
    entropy=0.0 #初始化香农熵
    for key in num_labels:
        prob=float(num_labels[key])/num_entropy #选择该标签的概率
        entropy-=prob*log(prob,2) #公式计算
    return entropy

def create_dataset():
    dataset=[[1,1,'yes'],
             [1,1,'yes'],
             [1,0,'no'],
             [0,1,'no'],
             [0,1,'no']]
    labels=['no surfacing','flippers']
    return dataset,labels

def split_dataset(dataset,axis,value):
```

```
    new_dataset=[] #创建新列表以存放满足要求的样本
    for feature_vector in dataset:
        if feature_vector[axis]==value:
            #下面这两句用来将axis特征去掉，并将符合条件的添加到返回的数据集
                                                    中
            reduced_feature_vector=feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis+1:])
            new_dataset.append(reduced_feature_vector)
    return new_dataset

"""
ID3
"""
def choose_best_feature_to_split(dataset):
    number_features=len(dataset[0])-1 #获取样本集中特征个数，-1是因为最后一
                                            列是label
    base_entropy = calculate_entropy(dataset)  # 计算根节点的信息熵
    best_info_gain = 0.0  # 初始化信息增益
    best_feature = -1  # 初始化最优特征的索引值
    for i in range(number_features):  # 遍历所有特征，i表示第几个特征
        #将dataset中的数据按行依次放入example中，然后取得example中的example
                                                [i]元素，即获得特征i的所有取
                                                值
        feature_list = [example[i] for example in dataset]
        #由上一步得到了特征i的取值，比如[1,1,1,0,0]，使用集合这个数据类型删
                                                除多余重复的取值，则剩下[1,0]
        unique_vals = set(feature_list) # 获取无重复的属性特征值
        new_entropy=0.0
        for value in unique_vals:
            sub_dataset =split_dataset(dataset, i, value) #逐个划分数据集，
                                                得到基于特征i和对应的取值
                                                划分后的子集
            prob = len(sub_dataset)/float(len(dataset)) #根据特征i可能取值
                                                划分出来的子集的概率
            new_entropy += prob * calculate_entropy(sub_dataset) #求解分支
                                                节点的信息熵
        info_gain = base_entropy - new_entropy  #计算信息增益，g(D,A)=H(D)-
                                            H(D|A)
        if (info_gain > best_info_gain):  # 对循环求得的信息增益进行大小比
```

```
                                                    较
            best_info_gain=info_gain
            best_feature = i #如果计算所得信息增益最大，则求得最佳划分方法
    return best_feature #返回划分属性（特征）


"""
C4.5
"""
def choose_best_feature_to_split_2(dataset):
    """
    按照最大信息增益比划分数据
    :param dataset: 样本数据，如：  [[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, '
                                                no'], [0, 1, 'no'], [0, 1, 'no']]
    :return:
    """
    number_feature = len(dataset[0]) - 1 # 特征个数，如：不浮出水面是否可以
                                                生存 和是否有脚蹼
    base_entropy = calculate_entropy(dataset) # 经验熵H(D)
    best_info_gain = 0.0
    best_feature = -1
    for i in range(number_feature):
        feature_list = [number[i] for number in dataset]  # 得到某个特征下
                                                所有值（某列）
        unique_feature_list = set(feature_list)  # 获取无重复的属性特征值
        new_entropy = 0
        split_info = 0.0
        for value in unique_feature_list:
            sub_dataset = split_dataset(dataset,i, value)
            prob = len(sub_dataset) / float(len(dataset))  # 即p(t)
            new_entropy += prob * calculate_entropy(sub_dataset)  # 对各子
                                                    集香农熵求和
            split_info += -prob * log(prob, 2)
        info_gain = base_entropy - new_entropy  # 计算信息增益，g(D,A)=H(D)
                                                -H(D|A)
        if split_info == 0:  # fix the overflow bug
            continue
        info_gain = info_gain / split_info
        # 最大信息增益比
        if info_gain > best_info_gain:
```

```
            best_info_gain = info_gain
            best_feature = i
    return best_feature

def majority_cnt(class_list):
    class_count={}
    for vote in class_list:
        if vote not in class_count.keys(): class_count[vote] = 0
        class_count[vote] += 1
    #分解为元组列表，operator.itemgetter(1)按照第二个元素的次序对元组进行排
                                          序，reverse=True是逆序，即按照从
                                          大到小的顺序排列
    sorted_class_count = sorted(class_count.iteritems(), key=operator.
                                          itemgetter(1), reverse=True)
    return sorted_class_count[0][0]

def create_decision_tree(dataset,labels):
    class_list = [example[-1] for example in dataset] #获取类别标签
    if class_list.count(class_list[0]) == len(class_list):
        return class_list[0] #类别完全相同则停止继续划分
    if len(dataset[0]) == 1:
        return majority_cnt(class_list) #遍历完所有特征时返回出现次数最多的
                                          类别
    best_feature = choose_best_feature_to_split(dataset) #选取最优划分特征
    best_feature_label = labels[best_feature] #获取最优划分特征对应的属性标
                                          签
    my_tree = {best_feature_label:{}} #存储树的所有信息
    del(labels[best_feature]) #删除已经使用过的属性标签
    feature_values = [example[best_feature] for example in dataset] #得到训
                                          练集中所有最优特征的属性值
    unique_vals = set(feature_values) #去掉重复的属性值
    for value in unique_vals: #遍历特征，创建决策树
        sub_labels = labels[:] #剩余的属性标签列表
        my_tree[best_feature_label][value] = create_decision_tree(
                                              split_dataset(dataset,
                                              best_feature, value),
                                              sub_labels) #递归函数实现决策
                                              树的构建
    return my_tree
```

```
def classify(input_tree,feature_labels,test_vector):
    first_string = list(input_tree.keys())[0] #获取根节点
    second_dict = input_tree[first_string] #获取下一级分支
    feature_index = feature_labels.index(first_string) #查找当前列表中第一
                                            个匹配firstStr变量的元素的索引
    key = test_vector[feature_index] #获取测试样本中，与根节点特征对应的取
                                            值
    value_feature = second_dict[key]#获取测试样本通过第一个特征分类器后的输
                                            出
    if isinstance(value_feature, dict): #判断节点是否为字典来以此判断是否为
                                            叶节点
        class_label = classify(value_feature, feature_labels, test_vector)
    else:
        class_label = value_feature#如果到达叶子节点，则返回当前节点的分类
                                                标签
    return class_label

"""
函数说明:绘制结点
Parameters:
    node_txt - 结点名
    center_pt - 文本位置
    parent_pt - 标注的箭头位置
    node_type - 结点格式
Returns:
    无
"""
def plot_node(node_txt, center_pt, parent_pt, node_type):
    arrow_args = dict(arrowstyle="<-") #定义箭头格式
    # 绘制结点
    create_plot.ax1.annotate(node_txt, xy=parent_pt, xycoords='axes
                                            fraction',xytext=center_pt,
                                            textcoords='axes fraction',va="
                                            center", ha="center", bbox=
                                            node_type, arrowprops=arrow_args)
"""
函数说明:获取决策树叶子结点的数目
Parameters:
```

```
    myTree - 决策树
Returns:
    numLeafs - 决策树的叶子结点的数目
"""
def get_num_leafs(my_tree):
    num_leafs = 0 #初始化叶子
    first_str = list(my_tree.keys())[0] ##python3中my——tree.keys()返回的
                                          是dict_keys,不在是list,所以不能使
                                          用myTree.keys()[0]的方法获取结点
                                          属性，可以使用list(myTree.keys())
                                          [0]
    second_dict = my_tree[first_str] ##获取下一组字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            num_leafs += get_num_leafs(second_dict[key])
        else:
            num_leafs += 1
    return num_leafs


"""
函数说明:获取决策树的层数
Parameters:
    myTree - 决策树
Returns:
    maxDepth - 决策树的层数
"""
def get_tree_depth(my_tree):
    max_depth = 0 #初始化决策树深度
    first_str = list(my_tree.keys())[0]
    second_dict = my_tree[first_str] #获取下一个字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            thisDepth = get_tree_depth(second_dict[key]) + 1
        else:
            thisDepth = 1
```

```
        if thisDepth > max_depth: #更新层数
            max_depth = thisDepth
    return max_depth

"""
函数说明:标注有向边属性值
Parameters:
    cntr_pt、parent_pt - 用于计算标注位置
    txt_string - 标注的内容
Returns:
    无
"""
def plot_mid_text(cntr_pt, parent_pt, txt_string):
    x_mid = (parent_pt[0] - cntr_pt[0]) / 2.0 + cntr_pt[0]
    y_mid = (parent_pt[1] - cntr_pt[1]) / 2.0 + cntr_pt[1]
    create_plot.ax1.text(x_mid, y_mid, txt_string)

"""
函数说明:绘制决策树
Parameters:
    my_tree - 决策树(字典)
    parent_pt - 标注的内容
    node_txt - 结点名
Returns:
    无
"""

leaf_node = dict(boxstyle="round4", fc="0.8")
decision_node = dict(boxstyle="sawtooth", fc="0.8")

def plot_tree(my_tree, parent_pt, node_txt):
    num_leafs = get_num_leafs(my_tree)
    depth = get_tree_depth(my_tree)
    first_str = list(my_tree.keys())[0]
    cntr_pt = (plot_tree.x_off + (1.0 + float(num_leafs)) / 2.0 / plot_tree
                                            .total_w, plot_tree.y_off)
    plot_mid_text(cntr_pt, parent_pt, node_txt)
    plot_node(first_str, cntr_pt, parent_pt, decision_node)
    second_dict = my_tree[first_str]
```

```
    plot_tree.y_off = plot_tree.y_off - 1.0 / plot_tree.total_d
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
            plot_tree(second_dict[key], cntr_pt, str(key))
        else:
            plot_tree.x_off = plot_tree.x_off + 1.0 / plot_tree.total_w
            plot_node(second_dict[key], (plot_tree.x_off, plot_tree.y_off),
                                                    cntr_pt, leaf_node)
            plot_mid_text((plot_tree.x_off, plot_tree.y_off), cntr_pt, str(
                                                   key))
    plot_tree.y_off = plot_tree.y_off + 1.0 / plot_tree.total_d

"""
函数说明:创建绘制面板
Parameters:
    in_tree - 决策树(字典)
Returns:
    无
"""
def create_plot(in_tree):
    fig = plt.figure(1,figsize=(12,8),facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    create_plot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plot_tree.total_w = float(get_num_leafs(in_tree))
    plot_tree.total_d = float(get_tree_depth(in_tree))
    plot_tree.x_off = -0.5 / plot_tree.total_w
    plot_tree.y_off = 1.0
    plot_tree(in_tree, (0.5, 1.0), '')
    plt.show()

"""
函数说明:存储决策树
Parameters:
    inputTree - 已经生成的决策树
    filename - 决策树的存储文件名
Returns:
    无
"""
```

```
def store_tree(input_tree,file_name):
    import pickle
    with open(file_name,'wb+') as fw:
        pickle.dump(input_tree,fw)

"""
函数说明:读取决策树
Parameters:
    file_name - 决策树的存储文件名
Returns:
    pickle.load(fr) - 决策树字典
"""
def grab_tree(file_name):
    import pickle
    fr = open(file_name, 'rb')
    return pickle.load(fr)
if __name__=='__main__':
    my_dataset,labels=create_dataset()
    print('=='*30)
    print('my_dataset=',my_dataset)
    print('labels=',labels)
    entropy = calculate_entropy(my_dataset)
    print('entropy=', entropy)
    """
    print(type(my_dataset))
    print('=='*30)
    a=[1,1,0]
    b=a[:0]
    print('b=',b)
    c=a[:2]
    print('c=',c)
    c.extend(a[3:])
    print('c.extend(a[3:])=',c.extend(a[3:]))
    print('=='*30)
    split_dataset = split_dataset(my_dataset, 0, 0)
    print('split_dataset=', split_dataset)
    print('=='*30)
    """
    print('=='*30)
```

```
selection_1=choose_best_feature_to_split(my_dataset)
print('selection_1=',selection_1)
print('=='*30)
selection_2=choose_best_feature_to_split_2(my_dataset)
print('selection_2=',selection_2)
print('=='*30)
my_tree=create_decision_tree(my_dataset,labels)
print('my_tree=',my_tree)
print('=='*30)
my_dataset, labels = create_dataset()
result=classify(my_tree,labels,[1,0])
print('result=',result)
create_plot(my_tree)
print('=='*30)
path='lenses.txt'
data=pd.read_csv(path,header=None)
print(data)
fr=open('lenses.txt')
lenses = [inst.strip().split('\t') for inst in fr.readlines()]
print('=='*30)
print('lenses=',lenses)
lensesLabels = ['age', 'prescript', 'astigmatic', 'tearRate']
lenses_tree = create_decision_tree(lenses, lensesLabels)
print('=='*30)
create_plot(lenses_tree)
store_tree(my_tree,'decision_tree_practice.txt')
myTree = grab_tree('decision_tree_practice.txt')
print(myTree)
```