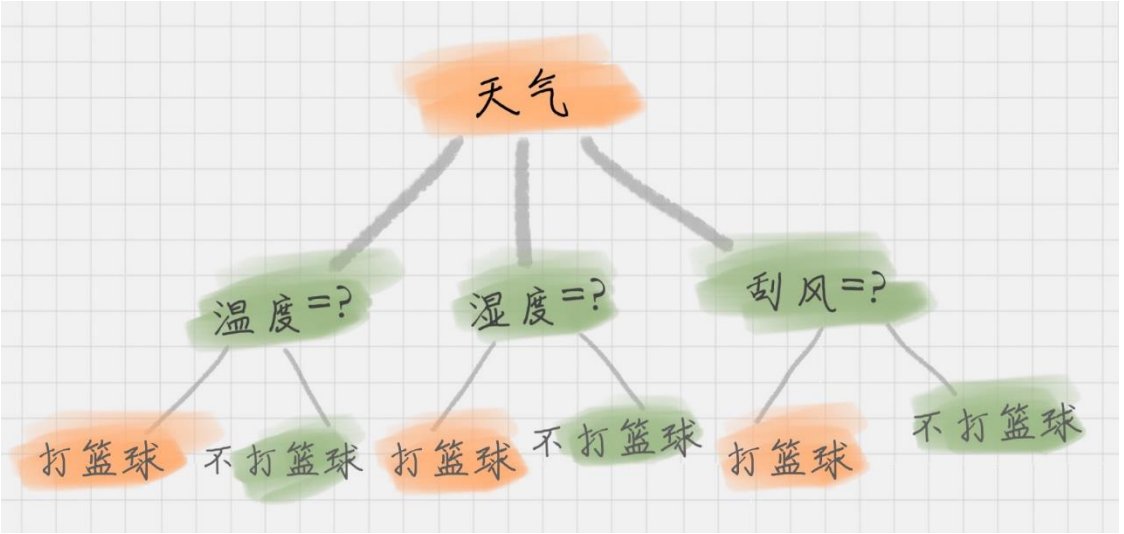
决策树

决策树

在现实生活中，我们会遇到各种选择，不论是选择男女朋友，还是挑选水果，都是基于以往的经验来做判断。如果把判断背后的逻辑整理成一个结构图，你会发现它实际上是一个树状图，这就是我们今天要讲的决策树。

决策树的工作原理

决策树基本上就是把我们以前的经验总结出来。如果我们要出门打篮球，一般会根据“天气”、“温度”、“湿度”、“刮风”这几个条件来判断，最后得到结果：去打篮球？还是不去？



上面这个图就是一棵典型的决策树。我们在做决策树的时候，会经历两个阶段：**构造**和**剪枝**。

构造

构造就是生成一棵完整的决策树。简单来说，**构造的过程就是选择什么属性作为节点的过程**，那么在构造过程中，会存在三种节点：

1. 根节点：就是树的最顶端，最开始的那个节点。在上图中，“天气”就是一个根节点；
2. 内部节点：就是树中间的那些节点，比如说“温度”、“湿度”、“刮风”；
3. 叶节点：就是树最底部的节点，也就是决策结果。

节点之间存在父子关系。比如根节点会有子节点，子节点会有子子节点，但是到了叶节点就停止了，叶节点不存在子节点。那么在构造过程中，你要解决三个重要的问题：

1. 选择哪个属性作为根节点；

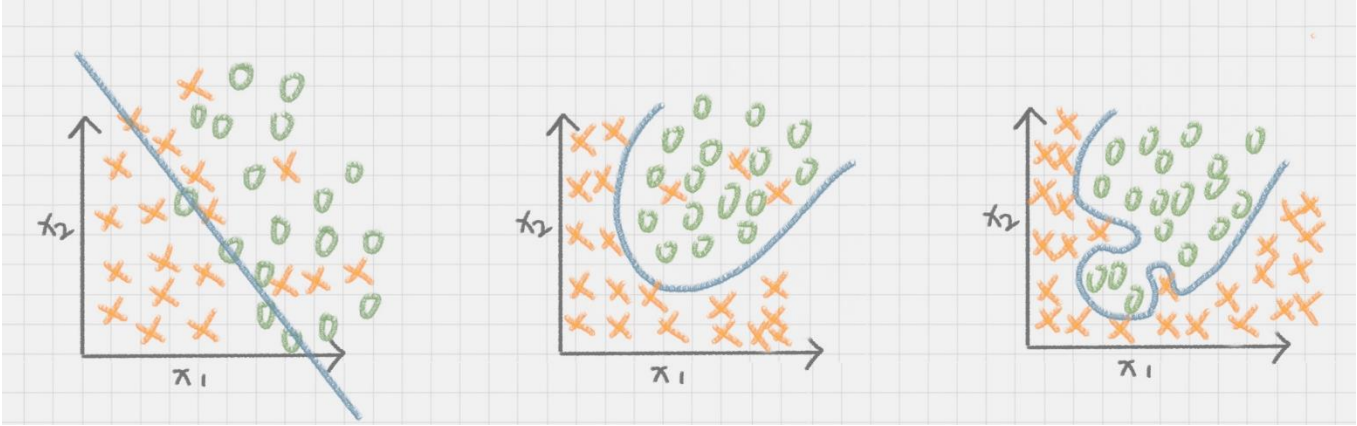
2. 选择哪些属性作为子节点；
3. 什么时候停止并得到目标状态，即叶节点。

剪枝

剪枝就是给决策树瘦身，这一步想实现的目标就是，不需要太多的判断，同样可以得到不错的结果。之所以这么做，是为了防止“过拟合”（Overfitting）现象的发生。

过拟合：指的是模型的训练结果“太好了”，以至于在实际应用的过程中，会存在“死板”的情况，导致分类错误。

欠拟合：指的是模型的训练结果不理想。



造成过拟合的原因：

一是因为训练集中样本量较小。如果决策树选择的属性过多，构造出来的决策树一定能够“完美”地把训练集中的样本分类，但是这样就会把训练集中一些数据的特点当成所有数据的特点，但这个特点不一定是全部数据的特点，这就使得这个决策树在真实的数据分类中出现错误，也就是模型的“泛化能力”差。

泛化能力：指的分类器是通过训练集抽象出来的分类能力，你也可以理解是举一反三的能力。如果我们太依赖于训练集的数据，那么得到的决策树容错率就会比较低，泛化能力差。因为训练集只是全部数据的抽样，并不能体现全部数据的特点。

剪枝的方法：

- **预剪枝：**在决策树构造时就进行剪枝。方法是，在构造的过程中对节点进行评估，如果对某个节点进行划分，在验证集中不能带来准确性的提升，那么对这个节点进行划分就没有意义，这时就会把当前节点作为叶节点，不对其进行划分。
- **后剪枝：**在生成决策树之后再进行剪枝。通常会从决策树的叶节点开始，逐层向上对每个节点进行评估。如果剪掉这个节点子树，与保留该节点子树在分类准确性上差别不大，或者剪掉该节点子树，能在验证集中带来准确性的提升，那么就可以把该节点子树进行剪枝。方法是：用这个节点子树的叶子节点来替代该节点，类标记为这个节点子树中最频繁的那个类。

如何判断要不要去打篮球？

天气	温度	湿度	刮风	是否打篮球
晴天	高	中	否	否
晴天	高	中	是	否
阴天	高	高	否	是
小雨	高	高	否	是
小雨	低	高	否	否
晴天	中	中	是	是
阴天	中	高	是	否

我们该如何构造一个判断是否去打篮球的决策树呢？再回顾一下决策树的构造原理，在决策过程中有三个重要的问题：将哪个属性作为根节点？选择哪些属性作为后继节点？什么时候停止并得到目标值？

显然将哪个属性（天气、温度、湿度、刮风）作为根节点是个关键问题，在这里我们先介绍两个指标：**纯度**和**信息熵**。

纯度：

你可以把决策树的构造过程理解成为寻找纯净划分的过程。数学上，我们可以用纯度来表示，纯度换一种方式来解释就是**让目标变量的分歧最小**。

举个例子，假设有 3 个集合：

- 集合 1：6 次都去打篮球；
- 集合 2：4 次去打篮球，2 次不去打篮球；
- 集合 3：3 次去打篮球，3 次不去打篮球。

按照纯度指标来说，集合 1> 集合 2> 集合 3。因为集合 1 的分歧最小，集合 3 的分歧最大。

信息熵：表示信息的不确定度

在信息论中，随机离散事件出现的概率存在着不确定性。为了衡量这种信息的不确定性，信息学之父香农引入了信息熵的概念，并给出了计算信息熵的数学公式：

$$Entropy(t) = -\sum_{i=0}^{c-1} p(i \mid t) \log_2 p(i \mid t)$$

p(i|t) 代表了节点 t 为分类 i 的概率，其中 log2 为取以 2 为底的对数。这里我们不是来介绍公式的，而是说存在一种度量，它能帮我们反映出来这个信息的不确定度。**当不确定性越大时，它所包含的信息量也就越大，信息熵也就越高。**

举个例子，假设有 2 个集合：

- 集合 1：5 次去打篮球，1 次不去打篮球；

- 集合 2：3 次去打篮球，3 次不去打篮球。

在集合 1 中，有 6 次决策，其中打篮球是 5 次，不打篮球是 1 次。那么假设：类别 1 为“打篮球”，即次数为 5；类别 2 为“不打篮球”，即次数为 1。那么节点划分为类别 1 的概率是 5/6，为类别 2 的概率是 1/6，带入上述信息熵公式可以计算得出：

$$Entropy(t) = -(1/6)\log_2(1/6) - (5/6)\log_2(5/6) = 0.65$$

同样，集合 2 中，也是一共 6 次决策，其中类别 1 中“打篮球”的次数是 3，类别 2“不打篮球”的次数也是 3，那么信息熵为多少呢？我们可以计算得出：

$$Entropy(t) = -(3/6)\log_2(3/6) - (3/6)\log_2(3/6) = 1$$

从上面的计算结果中可以看出，**信息熵越大，纯度越低。当集合中的所有样本均匀混合时，信息熵最大，纯度最低。**

我们在构造决策树的时候，会基于纯度来构建。而经典的 “不纯度”的指标有三种，分别是**信息增益（ID3 算法）、信息增益率（C4.5 算法）以及基尼指数（Cart 算法）**。

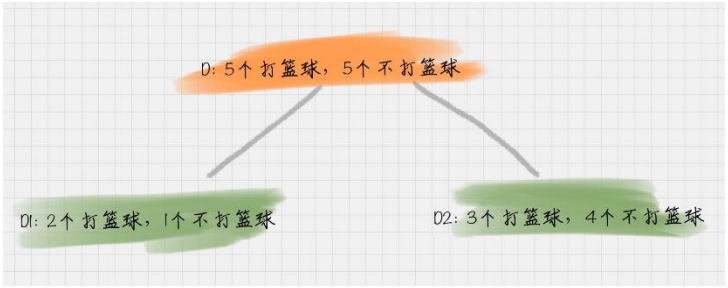
信息增益：

信息增益指的就是划分可以带来纯度的提高，信息熵的下降。它的计算公式，是**父亲节点的信息熵减去所有子节点的信息熵**。在计算的过程中，我们会计算每个子节点的归一化信息熵，即按照每个子节点在父节点中出现的概率，来计算这些子节点的信息熵。所以信息增益的公式可以表示为：

$$Gain(D,a) = Entropy(D) - \sum_{i=1}^{k} \frac{|D_i|}{|D|} Entropy(D_i)$$

公式中 D 是父亲节点，Di 是子节点，Gain(D,a)中的 a 作为 D 节点的属性选择。

假设 D 天气 = 晴的时候，会有 5 次去打篮球，5 次不打篮球。其中 D1 刮风 = 是，有 2 次打篮球，1 次不打篮球。D2 刮风 = 否，有 3 次打篮球，4 次不打篮球。那么 a 代表节点的属性，即天气 = 晴。



针对图上这个例子，D 作为节点的信息增益为：

$$Gain(D,a) = Entropy(D) - \left(\frac{3}{10}Entropy(D_1) + \frac{7}{10}Entropy(D_2)\right)$$

也就是 D 节点的信息熵 -2 个子节点的归一化信息熵。2 个子节点归一化信息熵 =3/10 的 D1 信息熵 +7/10 的 D2 信息熵。

我们基于 ID3 的算法规则，完整地计算下我们的训练集，训练集中一共有 7 条数据，3 个打篮球，4 个不打篮球，所以根节点的信息熵是：

$$Ent(D) = -\sum_{k=1}^{2} p_k \log_2 p_k = -(\frac{3}{7}\log_2 \frac{3}{7} + \frac{4}{7}\log_2 \frac{4}{7}) = 0.985$$

如果你将天气作为属性的划分，会有三个叶子节点 D1、D2 和 D3，分别对应的是晴天、阴天和小雨。我们用 + 代表去打篮球，- 代表不去打篮球。那么第一条记录，晴天不去打篮球，可以记为 1-，于是我们可以用下面的方式来记录 D1，D2，D3：

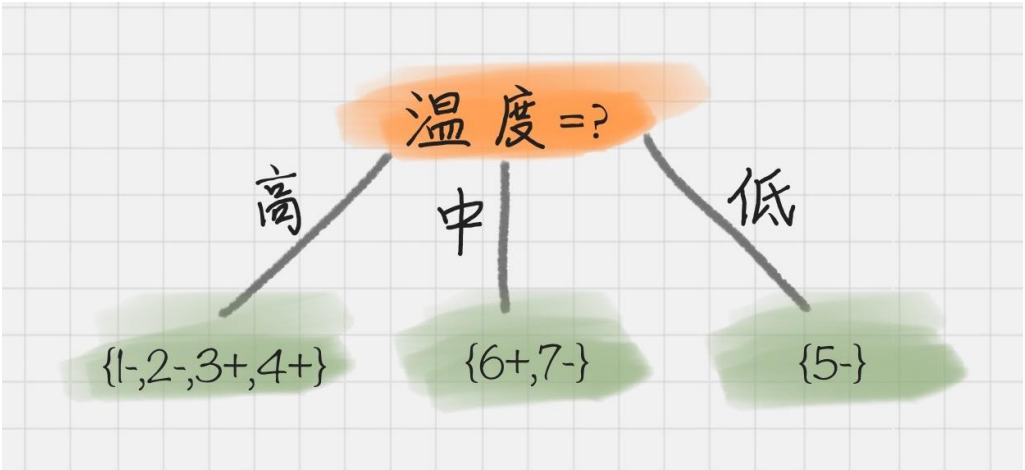
D1(天气 = 晴天)={1-,2-,6+}
D2(天气 = 阴天)={3+,7-}
D3(天气 = 小雨)={4+,5-}
我们先分别计算三个叶子节点的信息熵：

$$\mathrm{Ent}(D_1) = -\left(\frac{1}{3}\log_2\frac{1}{3} + \frac{2}{3}\log_2\frac{2}{3}\right) = 0.918$$

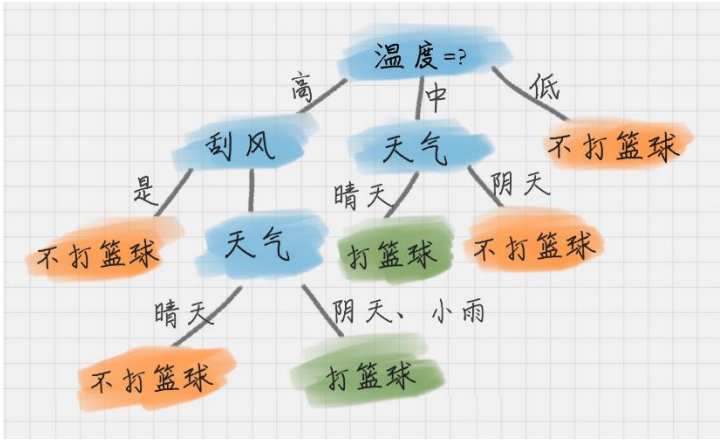
$$\mathrm{Ent}(D_2) = -\left(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2}\right) = 1.0$$

$$\mathrm{Ent}(D_3) = -\left(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2}\right) = 1.0$$

因为 D1 有 3 个记录，D2 有 2 个记录，D3 有 2 个记录，所以 D 中的记录一共是 3+2+2=7，即总数为 7。所以 D1 在 D（父节点）中的概率是 3/7，D2 在父节点的概率是 2/7，D3 在父节点的概率是 2/7。那么作为子节点的归一化信息熵 = 3/7*0.918+2/7*1.0=0.965。
因为我们用 ID3 中的信息增益来构造决策树，所以要计算每个节点的信息增益。
天气作为属性节点的信息增益为，Gain(D，天气)=0.985-0.965=0.020。
同理我们可以计算出其他属性作为根节点的信息增益，它们分别为：
Gain(D，温度)=0.128
Gain(D，湿度)=0.020
Gain(D，刮风)=0.020
我们能看出来温度作为属性的信息增益最大。因为 ID3 就是要将信息增益最大的节点作为父节点，这样可以得到纯度高的决策树，所以我们将温度作为根节点。其决策树状图分裂为下图所示：



然后我们要将上图中第一个叶节点，也就是 D1={1-,2-,3+,4+}进一步进行分裂，往下划分，计算其不同属性（天气、湿度、刮风）作为节点的信息增益，可以得到：
Gain(D，天气)=0
Gain(D，湿度)=0
Gain(D，刮风)=0.0615
我们能看到刮风为 D1 的节点都可以得到最大的信息增益，这里我们选取刮风作为节点。同理，我们可以按照上面的计算步骤得到完整的决策树，结果如下：



于是我们通过 ID3 算法得到了一棵决策树。ID3 的算法规则相对简单，可解释性强。同样也存在缺陷，比如我们会发现 **ID3 算法倾向于选择取值比较多的属性**。这样，如果我们把"编号"作为一个属性（一般情况下不会这么做，这里只是举个例子），那么"编号"将会被选为最优属性 。但实际上"编号"是无关属性的，它对"打篮球"的分类并没有太大作用。

所以 ID3 有一个缺陷就是，有些属性可能对分类任务没有太大作用，但是他们仍然可能会被选为最优属性。这种缺陷不是每次都会发生，只是存在一定的概率。在大部分情况下，ID3 都能生成不错的决策树分类。针对可能发生的缺陷，后人提出了新的算法进行改进。

在 ID3 算法上进行改进的 C4.5 算法

1. 采用信息增益率

因为 ID3 在计算的时候，倾向于选择取值多的属性。为了避免这个问题，C4.5 采用信息增益率的方式来选择属性。**信息增益率 = 信息增益 / 属性熵**

当属性有很多值的时候，相当于被划分成了许多份，虽然信息增益变大了，但是对于 C4.5 来说，属性熵也会变大，所以整体的信息增益率并不大。

2. 采用悲观剪枝

ID3 构造决策树的时候，容易产生过拟合的情况。在 C4.5 中，会在决策树构造之后采用悲观剪枝（PEP），这样可以提升决策树的泛化能力。

悲观剪枝是后剪枝技术中的一种，通过递归估算每个内部节点的分类错误率，比较剪枝前后这个节点的分类错误率来决定是否对其进行剪枝。这种剪枝方法不再需要一个单独的测试数据集。

3. 离散化处理连续属性

C4.5 可以处理连续属性的情况，对连续的属性进行离散化的处理。比如打篮球存在的"湿度"属性，不按照"高、中"划分，而是按照湿度值进行计算，那么湿度取什么值都有可能。该怎么选择这个阈值呢，**C4.5 选择具有最高信息增益的划分所对应的阈值**。

4. 处理缺失值

针对数据集不完整的情况，C4.5 也可以进行处理。

假如我们得到的是如下的数据，你会发现这个数据中存在两点问题。第一个问题是，数据集中存在数值缺失的情况，如何进行属性选择？第二个问题是，假设已经做了属性划分，但是样本在这个属性上有缺失值，该如何对样本进行划分？

ID	天气	温度	湿度	刮风	是否打篮球
1	晴天	-	中	否	否
2	晴天	高	中	是	否
3	阴天	高	高	否	是
4	小雨	高	高	否	是
5	小雨	低	高	否	否
6	晴天	中	中	是	是
7	阴天	中	高	是	否

我们不考虑缺失的数值，可以得到温度 D={2-,3+,4+,5-,6+,7-}。温度 = 高：D1={2-,3+,4+}；温度 = 中：D2={6+,7-}；温度 = 低：D3={5-} 。这里 + 号代表打篮球，- 号代表不打篮球。比如 ID=2 时，决策是不打篮球，我们可以记录为 2-。
所以三个叶节点的信息熵可以结算为：

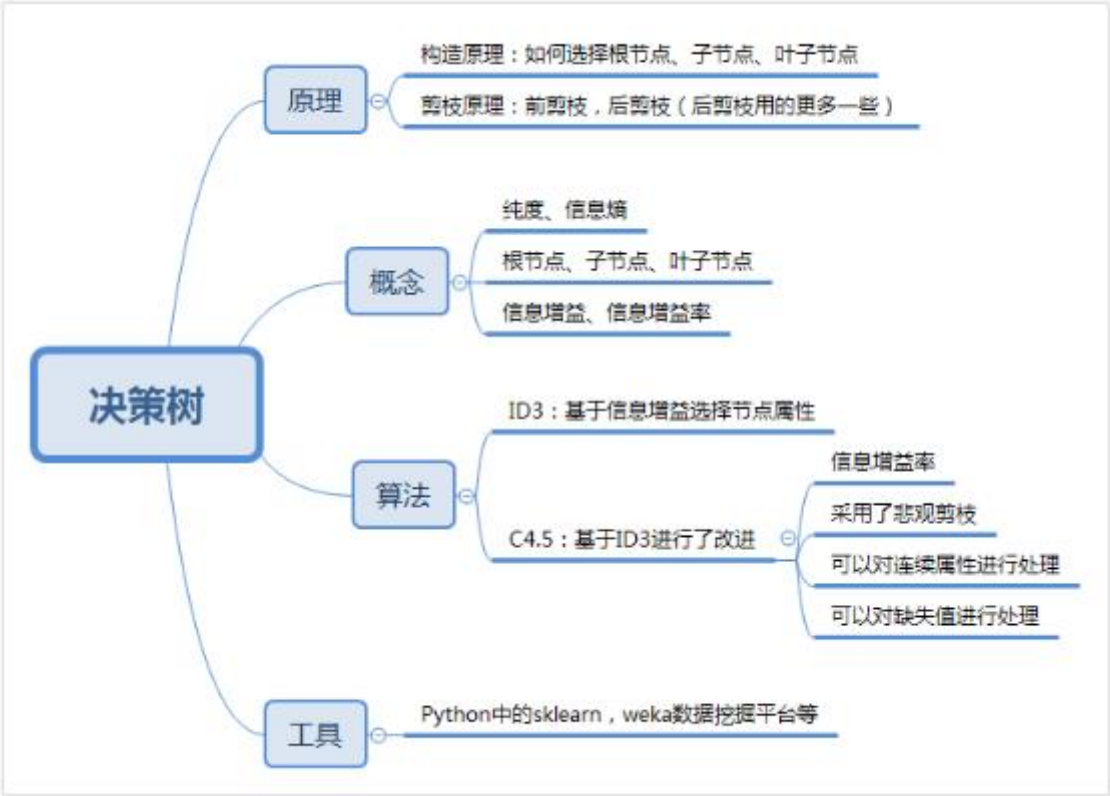
$$Ent(D_1) = -(\frac{1}{3}\log_2\frac{1}{3} + \frac{2}{3}\log_2\frac{2}{3}) = 0.918$$

$$Ent(D_2) = -(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2}) = 1.0$$

$$Ent(D_3) = 0$$

这三个节点的归一化信息熵为 3/6*0.918+2/6*1.0+1/6*0=0.792。
针对将属性选择为温度的信息增益率为：
Gain(D', 温度)=Ent(D')-0.792=1.0-0.792=0.208
D'的样本个数为 6，而 D 的样本个数为 7，所以所占权重比例为 6/7，所以 Gain(D'，温度) 所占权重比例为 6/7，所以：
Gain(D, 温度)=6/7*0.208=0.178
这样即使在温度属性的数值有缺失的情况下，我们依然可以计算信息增益，并对属性进行选择。
小结：

首先 ID3 算法的优点是方法简单，缺点是对噪声敏感。训练数据如果有少量错误，可能会产生决策树分类错误。C4.5 在 IID3 的基础上，用信息增益率代替了信息增益，解决了噪声敏感的问题，并且可以对构造树进行剪枝、处理连续数值以及数值缺失等情况，但是由于 C4.5 需要对数据集进行多次扫描，算法效率相对较低。



机器学习实战笔记(Python 实现)-02-决策树

目录

- 1、算法概述及实现
  - 1.1 算法理论
  - 1.2 构造决策树
- 2、测试分类和存储分类器
- 3、使用 Matplotlib 绘制树形图
  - 3.1 绘制树节点
  - 3.2 构造注解树

- 4、实例：使用决策树预测隐形眼镜类型
  - 4.1 处理流程
  -
  - 4.2 Python 实现代码

正文

本系列文章为《机器学习实战》学习笔记，内容整理自书本，网络以及自己的理解，如有错误欢迎指正。

源码在 Python3.5 上测试均通过，代码及数据 --> https://github.com/Wellat/MLaction

---

1、算法概述及实现

1.1 算法理论

决策树构建的伪代码如下：

输入: 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
　　属性集 $A = \{a_1, a_2, \ldots, a_d\}$.
过程: 函数 TreeGenerate(D, A)
1: 生成结点 node;
2: **if** D 中样本全属于同一类别 C **then**
3: 　将 node 标记为 C 类叶结点; **return**
4: **end if**
5: **if** $A = \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
6: 　将 node 标记为叶结点, 其类别标记为 D 中样本数最多的类; **return**
7: **end if**
8: 从 A 中选择最优划分属性 a_*;
9: **for** a_* 的每一个值 a_*^v **do**
10: 　为 node 生成一个分支; 令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集;
11: 　**if** D_v 为空 **then**
12: 　　将分支结点标记为叶结点, 其类别标记为 D 中样本最多的类; **return**
13: 　**else**
14: 　　以 TreeGenerate(D_v, $A \setminus \{a_*\}$)为分支结点
15: 　**end if**
16: **end for**
输出: 以 node 为根结点的一棵决策树

显然，决策树的生成是一个递归过程，在决策树基本算法中，有 3 种情导致递归返回：(1)当前结点包含的样本全属于同一类别，无需划分；(2)属性集为空，或是所有样本在所有属性上取值相同，无法划分；(3)当前结点包含的样本集合为空，不能划分。

划分选择

决策树学习的关键是第 8 行，即如何选择最优划分属性，一般而言，随着划分过程不断进行，我们希望决策树的分支节点所包含的样本尽可能属于同一类别，即结点的"纯度"越来越高。

①ID3 算法

ID3 算法通过对比选择不同特征下数据集的**信息增益**和**香农熵**来确定最优划分特征。

香农熵：

$$H(U) = E[-\log p_i] = -\sum_{i=1}^{n} p_i \log p_i .$$

```
from collections import Counter
import operator
import math

def calcEnt(dataSet):
    classCount = Counter(sample[-1] for sample in dataSet)
    prob = [float(v)/sum(classCount.values()) for v in classCount.values()]
    return reduce(operator.add, map(lambda x: -x*math.log(x, 2), prob))
```

纯度差，也称为**信息增益**，表示为

$$\Delta=\mathrm{I(parent)}-\sum_{j=1}^{k}\frac{N(v_j)}{N} * I(v_j)$$

上面公式实际上就是当前节点的不纯度减去子节点不纯度的加权平均数，权重由子节点记录数与当前节点记录数的比例决定。信息增益越大，则意味着使用属性 a 来进行划分所获得的"纯度提升"越大，效果越好。

信息增益准则对可取值数目较多的属性有所偏好。

②C4.5 算法

C4.5 决策树生成算法相对于 ID3 算法的重要改进是使用信息增益率来选择节点属性。它克服了 ID3 算法存在的不足：ID3 算法只适用于离散的描述属性，对于连续数据需离散化，而 C4.5 则离散连续均能处理。

增益率定义为：

$$\text{Gain\_ratio}(D,a)=\frac{\text{Gain}(D,a)}{\text{IV}(a)}$$

其中

$$\text{IV}(a)=-\sum_{v=1}^{V}\frac{|D^v|}{|D|}\log_2\frac{|D^v|}{|D|}$$

需注意的是，增益率准则对可取值数目较少的属性有所偏好，因此，C4.5 算法并不是直接选择增益率最大的候选划分属性，而是使用了一个启发式：先从候选划分属性中找出信息增益高于平均水平的属性，再从中选择增益率最高的。

③CART 算法

CART 决策树使用"基尼指数"来选择划分属性，数据集 D 的纯度可用基尼值来度量：

$$\begin{aligned}\text{Gini}(D)&=\sum_{k=1}^{|\mathcal{Y}|}\sum_{k'\neq k}p_k p_{k'}\\&=1-\sum_{k=1}^{|\mathcal{Y}|}p_k^2\ .\end{aligned}$$

它反映了从数据集 D 中随机抽取两个样本，其类别标记不一致的概率。**因此 Gini(D)越小，则数据集 D 的纯度越高**。

```
from  collections import Counter
import operator

def calcGini(dataSet):
    labelCounts = Counter(sample[-1] for sample in dataSet)
    prob = [float(v)/sum(labelCounts.values()) for v in labelCounts.values()]
    return 1 - reduce(operator.add, map(lambda x: x**2, prob))
```

剪枝处理

为避免过拟合，需要对生成树剪枝。决策树剪枝的基本策略有“**预剪枝**”和“**后剪枝**”。预剪枝是指在决策树生成过程中，对每个结点划分前先进行估计，若当前结点的划分不能带来决策树**泛化性能提升**，则停止划分并将当前结点标记为叶结点（有欠拟合风险）。后剪枝则是先从训练集生成一棵完整的决策树，然后自底向上地对非叶结点进行考察，若将该结点对应的子树替换为叶结点能带来决策树泛化性能提升，则将该子树替换为叶节点。

1.2 构造决策树

本书使用 ID3 算法划分数据集，即通过对比选择不同特征下数据集的信息增益和香农熵来确定最优划分特征。

1.2.1 计算香农熵：

```
from math import log
import operator

def createDataSet():
    '''
    产生测试数据
    '''
    dataSet = [[1, 1, 'yes'],
               [1, 1, 'yes'],
               [1, 0, 'no'],
               [0, 1, 'no'],
               [0, 1, 'no']]
    labels = ['no surfacing','flippers']
    return dataSet, labels

def calcShannonEnt(dataSet):
    '''
    计算给定数据集的香农熵
    '''
    numEntries = len(dataSet)
    labelCounts = {}
    #统计每个类别出现的次数，保存在字典 labelCounts 中
    for featVec in dataSet:
        currentLabel = featVec[-1]
        if currentLabel not in labelCounts.keys(): labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1 #如果当前键值不存在，则扩展字典并将当前键值加入字典
    shannonEnt = 0.0
    for key in labelCounts:
        #使用所有类标签的发生频率计算类别出现的概率
```

```
        prob = float(labelCounts[key])/numEntries
        #用这个概率计算香农熵
        shannonEnt -= prob * log(prob,2) #取 2 为底的对数
    return shannonEnt

if __name__ == "__main__":
    '''
    计算给定数据集的香农熵
    '''
    dataSet,labels = createDataSet()
    shannonEnt = calcShannonEnt(dataSet)
```

1.2.2 划分数据集

```
def splitDataSet(dataSet, axis, value):
    '''
    按照给定特征划分数据集
    dataSet：待划分的数据集
    axis：    划分数据集的第 axis 个特征
    value：   特征的返回值（比较值）
    '''
    retDataSet = []
    #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
    for featVec in dataSet:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
            #extend()和 append()方法功能相似，但在处理列表时，处理结果完全不同
            #a=[1,2,3]  b=[4,5,6]
            #a.append(b) = [1,2,3,[4,5,6]]
            #a.extend(b) = [1,2,3,4,5,6]
    return retDataSet
```

划分数据集的结果如下所示：

```
[[1, 'no'], [1, 'no']]
```

选择最好的数据集划分方式。接下来我们将遍历整个数据集，循环计算香农熵和 splitDataSet() 函数，找到最好的特征划分方式。

```
def chooseBestFeatureToSplit(dataSet):
```

```
    '''
    选择最好的数据集划分方式
    输入：数据集
    输出：最优分类的特征的 index
    '''
    #计算特征数量
    numFeatures = len(dataSet[0]) - 1
    baseEntropy = calcShannonEnt(dataSet)
    bestInfoGain = 0.0; bestFeature = -1
    for i in range(numFeatures):
        #创建唯一的分类标签列表
        featList = [example[i] for example in dataSet]
        uniqueVals = set(featList)
        #计算每种划分方式的信息熵
        newEntropy = 0.0
        for value in uniqueVals:
            subDataSet = splitDataSet(dataSet, i, value)
            prob = len(subDataSet)/float(len(dataSet))
            newEntropy += prob * calcShannonEnt(subDataSet)
        infoGain = baseEntropy - newEntropy
        #计算最好的信息增益，即 infoGain 越大划分效果越好
        if (infoGain > bestInfoGain):
            bestInfoGain = infoGain
            bestFeature = i
    return bestFeature
```

1.2.3 递归构建决策树

目前我们已经学习了从数据集构造决策树算法所需要的子功能模块，其工作原理如下：得到原始数据集，然后基于最好的属性值划分数据集，由于特征值可能多于两个，因此可能存在大于两个分支的数据集划分。第一次划分之后，数据将被向下传递到树分支的下一个节点，在这个节点上，我们可以再次划分数据。因此我们可以采用递归的原则处理数据集。递归结束的条件是：程序遍历完所有划分数据集的属性，或者每个分支下的所有实例都具有相同的分类。

由于特征数目并不是在每次划分数据分组时都减少，因此这些算法在实际使用时可能引起一定的问题。目前我们并不需要考虑这个问题，只需要在算法开始运行前计算列的数目，查看算法是否使用了所有属性即可。如果数据集已经处理了所有属性，但是类标签依然不是唯一的，此时我们通常会采用多数表决的方法决定该叶子节点的分类。

在 trees.py 中增加如下投票表决代码：

```
import operator
def majorityCnt(classList):
```

```
    '''
    投票表决函数
    输入 classList:标签集合，本例为：['yes', 'yes', 'no', 'no', 'no']
    输出：得票数最多的分类名称
    '''
    classCount={}
    for vote in classList:
        if vote not in classCount.keys(): classCount[vote] = 0
        classCount[vote] += 1
    #把分类结果进行排序，然后返回得票数最多的分类结果
    sortedClassCount = sorted(classCount.items(), key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]
```

创建树的函数代码（主函数）：

```
def createTree(dataSet,labels):
    '''
    创建树
    输入：数据集和标签列表
    输出：树的所有信息
    '''
    # classList 为数据集的所有类标签
    classList = [example[-1] for example in dataSet]
    # 停止条件 1:所有类标签完全相同，直接返回该类标签
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    # 停止条件 2:遍历完所有特征时仍不能将数据集划分成仅包含唯一类别的分组，则返回出现次数最多的类标签
    #
    if len(dataSet[0]) == 1:
        return majorityCnt(classList)
    # 选择最优分类特征
    bestFeat = chooseBestFeatureToSplit(dataSet)
    bestFeatLabel = labels[bestFeat]
    # myTree 存储树的所有信息
    myTree = {bestFeatLabel:{}}
    # 以下得到列表包含的所有属性值
    del(labels[bestFeat])
```

```
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    # 遍历当前选择特征包含的所有属性值
    for value in uniqueVals:
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value),subLabels)
    return myTree
```

本例返回 myTree 为字典类型，如下：

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

2、测试分类和存储分类器

利用决策树的分类函数：

```
def classify(inputTree,featLabels,testVec):
    '''
    决策树的分类函数
    inputTree:训练好的树信息
    featLabels:标签列表
    testVec:测试向量
    '''
    # 在 2.7 中，找到 key 所对应的第一个元素为：firstStr = myTree.keys()[0]，
    # 这在 3.4 中运行会报错：‘dict_keys ‘ object does not support indexing，这是因为 python3 改变了 dict.keys，
    # 返回的是 dict_keys 对象，支持 iterable 但不支持 indexable，
    # 我们可以将其明确的转化成 list，则此项功能在 3 中应这样实现：
    firstSides = list(inputTree.keys())
    firstStr = firstSides[0]
    secondDict = inputTree[firstStr]
    # 将标签字符串转换成索引
    featIndex = featLabels.index(firstStr)
    key = testVec[featIndex]
    valueOfFeat = secondDict[key]
    # 递归遍历整棵树，比较 testVec 变量中的值与树节点的值，如果到达叶子节点，则返回当前节点的分类标签
    if isinstance(valueOfFeat, dict):
```

```
            classLabel = classify(valueOfFeat, featLabels, testVec)
    else: classLabel = valueOfFeat
    return classLabel

if __name__== "__main__":
    '''
    测试分类效果
    '''
    dataSet,labels = createDataSet()
    myTree = createTree(dataSet,labels)
    ans = classify(myTree,labels,[1,0])
```

决策树模型的存储

```
def storeTree(inputTree,filename):
    '''
    使用 pickle 模块存储决策树
    '''
    import pickle
    fw = open(filename,'wb+')
    pickle.dump(inputTree,fw)
    fw.close()

def grabTree(filename):
    '''
    导入决策树模型
    '''
    import pickle
    fr = open(filename,'rb')
    return pickle.load(fr)

if __name__== "__main__":
    '''
    存取操作
    '''
```

```
    storeTree(myTree,'mt.txt')
    myTree2 = grabTree('mt.txt')
```

3、使用 Matplotlib 绘制树形图

上节我们已经学习如何从数据集中创建决策树，然而字典的表示形式非常不易于理解，决策树的主要优点就是直观易于理解，如果不能将其直观显示出来，就无法发挥其优势。本节使用 Matplotlib 库编写代码绘制决策树。

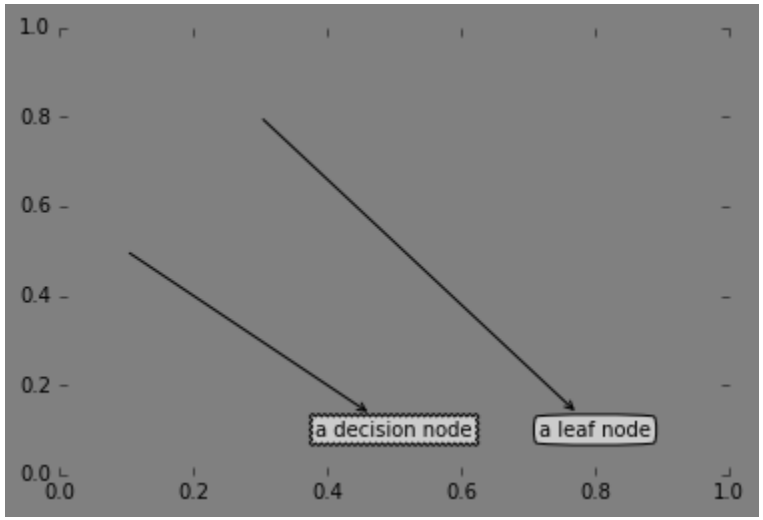
创建名为 treePlotter.py 的新文件：

3.1 绘制树节点

```
import matplotlib.pyplot as plt

# 定义文本框和箭头格式
decisionNode = dict(boxstyle="sawtooth", fc="0.8")
leafNode = dict(boxstyle="round4", fc="0.8")
arrow_args = dict(arrowstyle="<-")

# 绘制带箭头的注释
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    createPlot.ax1.annotate(nodeTxt, xy=parentPt,  xycoords='axes fraction',
             xytext=centerPt, textcoords='axes fraction',
             va="center", ha="center", bbox=nodeType, arrowprops=arrow_args )

def createPlot():
    fig = plt.figure(1, facecolor='grey')
    fig.clf()
    # 定义绘图区
    createPlot.ax1 = plt.subplot(111, frameon=False) #ticks for demo puropses
    plotNode('a decision node', (0.5, 0.1), (0.1, 0.5), decisionNode)
    plotNode('a leaf node', (0.8, 0.1), (0.3, 0.8), leafNode)
    plt.show()

if __name__== "__main__":
    '''
    绘制树节点
    '''
```

```
27      createPlot()
```

结果如下：



3.2 构造注解树

绘制一棵完整的树需要一些技巧。我们虽然有 x, y 坐标，但是如何放置所有的树节点却是个问题。我们必须知道有多少个叶节点，以便可以正确确 x 轴的长度；我们还需要知道树有多少层，来确定 y 轴的高度。这里另一两个新函数 getNumLeafs() 和 getTreeDepth()，来获取叶节点的数目和树的层数，createPlot() 为主函数，完整代码如下：

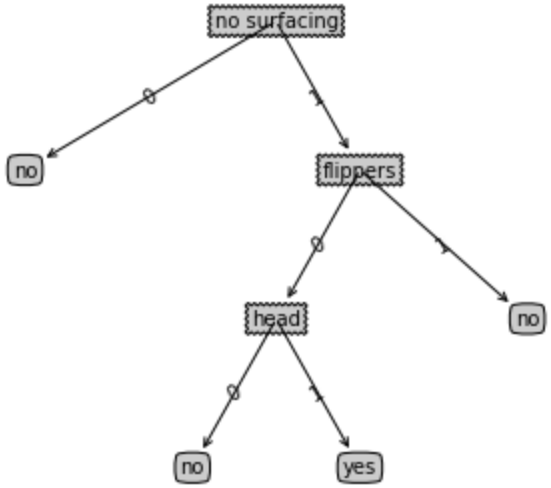
```
import matplotlib.pyplot as plt

# 定义文本框和箭头格式
decisionNode = dict(boxstyle="sawtooth", fc="0.8")
leafNode = dict(boxstyle="round4", fc="0.8")
arrow_args = dict(arrowstyle="<-")

# 绘制带箭头的注释
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    createPlot.ax1.annotate(nodeTxt, xy=parentPt,  xycoords='axes fraction',
             xytext=centerPt, textcoords='axes fraction',
             va="center", ha="center", bbox=nodeType, arrowprops=arrow_args )

def createPlot(inTree):
```

```
    '''
    绘树主函数
    '''
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    # 设置坐标轴数据
    axprops = dict(xticks=[], yticks=[])
    # 无坐标轴
    createPlot.ax1 = plt.subplot(111, frameon=False, **axprops)
    # 带坐标轴
#    createPlot.ax1 = plt.subplot(111, frameon=False)
    plotTree.totalW = float(getNumLeafs(inTree))
    plotTree.totalD = float(getTreeDepth(inTree))
    # 两个全局变量 plotTree.xOff 和 plotTree.yOff 追踪已经绘制的节点位置，
    # 以及放置下一个节点的恰当位置
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;
    plotTree(inTree, (0.5,1.0), '')
    plt.show()


def getNumLeafs(myTree):
    '''
    获取叶节点的数目
    '''
    numLeafs = 0
    firstSides = list(myTree.keys())
    firstStr = firstSides[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        # 判断节点是否为字典来以此判断是否为叶子节点
        if type(secondDict[key]).__name__=='dict':
            numLeafs += getNumLeafs(secondDict[key])
        else:   numLeafs +=1
    return numLeafs

```

```
50 def getTreeDepth(myTree):
51     '''
52     获取树的层数
53     '''
54     maxDepth = 0
55     firstSides = list(myTree.keys())
56     firstStr = firstSides[0]
57     secondDict = myTree[firstStr]
58     for key in secondDict.keys():
59         if type(secondDict[key]).__name__=='dict':
60             thisDepth = 1 + getTreeDepth(secondDict[key])
61         else:   thisDepth = 1
62         if thisDepth > maxDepth: maxDepth = thisDepth
63     return maxDepth
64
65
66 def plotMidText(cntrPt, parentPt, txtString):
67     '''
68     计算父节点和子节点的中间位置，并在此处添加简单的文本标签信息
69     '''
70     xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]
71     yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]
72     createPlot.ax1.text(xMid, yMid, txtString, va="center", ha="center", rotation=30)
73
74 def plotTree(myTree, parentPt, nodeTxt):#if the first key tells you what feat was split on
75     # 计算宽与高
76     numLeafs = getNumLeafs(myTree)  #this determines the x width of this tree
77     depth = getTreeDepth(myTree)
78     firstSides = list(myTree.keys())
79     firstStr = firstSides[0]
80     cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)
81     # 标记子节点属性值
82     plotMidText(cntrPt, parentPt, nodeTxt)
83     plotNode(firstStr, cntrPt, parentPt, decisionNode)
84     secondDict = myTree[firstStr]
```

```
85      # 减少 y 偏移
86      plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD
87      for key in secondDict.keys():
88          if type(secondDict[key]).__name__=='dict':#test to see if the nodes are dictonaires, if not they are leaf nodes
89              plotTree(secondDict[key],cntrPt,str(key))        #recursion
90          else:   #it's a leaf node print the leaf node
91              plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW
92              plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff), cntrPt, leafNode)
93              plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
94      plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD
95
96
97  def retrieveTree(i):
98      '''
99      保存了树的测试数据
100     '''
101     listOfTrees =[{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}},
102                   {'no surfacing': {0: 'no', 1: {'flippers': {0: {'head': {0: 'no', 1: 'yes'}}, 1: 'no'}}}}
103                   ]
104     return listOfTrees[i]
105
106
107
108 if __name__== "__main__":
109     '''
110     绘制树
111     '''
112     createPlot(retrieveTree(1))
```

测试结果：



4、实例：使用决策树预测隐形眼镜类型

4.1 处理流程

示例：使用决策树预测隐形眼镜类型

(1) 收集数据：提供的文本文件。
(2) 准备数据：解析tab键分隔的数据行。
(3) 分析数据：快速检查数据，确保正确地解析数据内容，使用createPlot()函数绘制最终的树形图。

(4) 训练算法：使用3.1节的createTree()函数。
(5) 测试算法：编写测试函数验证决策树可以正确分类给定的数据实例。
(6) 使用算法：存储树的数据结构，以便下次使用时无需重新构造树。

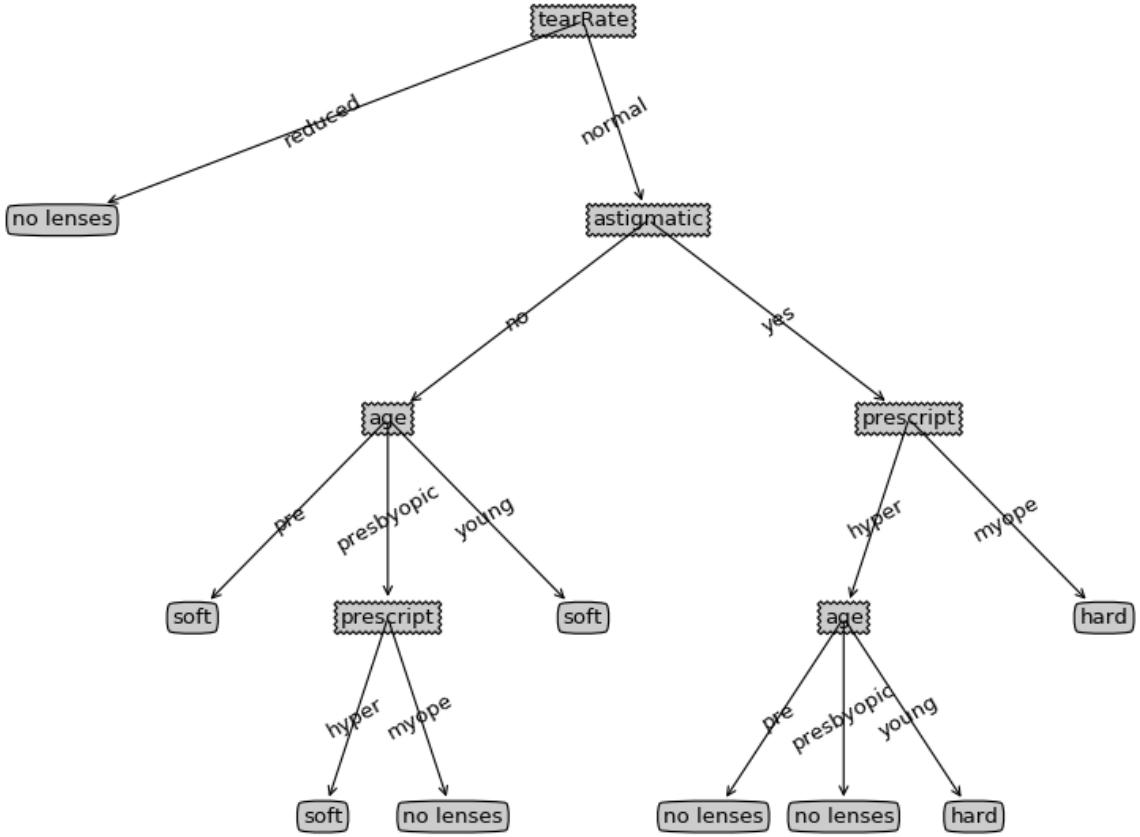
数据格式如下所示，其中最后一列表示类标签：

```
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
```

4.2 Python 实现代码

```python
import trees
import treePlotter

fr = open('lenses.txt')
lenses = [inst.strip().split('\t') for inst in fr.readlines()]
lensesLabels=['age','prescript','astigmatic','tearRate']
lensesTree = trees.createTree(lenses,lensesLabels)
treePlotter.createPlot(lensesTree)
```

产生的决策树：



本节使用的算法成为 ID3，它是一个号的算法但无法直接处理数值型数据，尽管我们可以通过量化的方法将数值型数据转化为标称型数值，但如果存在太多的特征划分，ID3 算法仍然会面临其他问题。