

第三章 决策树实战

3.1 决策树的基本概念

3.1.1 决策树的基本概念

决策树 (decision tree) 是一类常见的机器学习方法. 在二分类中, 学得一个模型用以对新示例进行分类, 整个把样本分类的任务, 可以堪称对"当前样本属于正类吗?"这个问题的"决策"或"判定"过程. 如在对"这是好瓜吗?"这个问题决策时, 通常会进行一系列的判断或"子决策".

决策过程的最终结论对应了我们所希望的判定结果, 如"是"或"不是"好瓜; 决策过程中提出的每个判定问题都是对某个属性的"测试"; 每个测试的结果或是导出最终结论, 或是导出进一步的判定问题(其考虑的范围是上次决策结果的限定范围之内).

3.1.2 根结点, 叶结点和内部结点

1 三类结点的概念

1. 根结点: 第一个判定的集合, 根结点包含样本全集
2. 叶结点: 叶结点对应于决策结果
3. 内部结点: 中间的结点, 需要进一步进行属性测试的结点

总结:

1. 一般的, 一棵决策树包含一个根结点、若干个内部结点和若干个叶结点;
2. 叶结点对应于决策结果, 其他每个结点则对应于一个属性测试(包括根结点和内部结点)
3. 每个结点包含的样本集合根据属性测试的结果被划分到子结点中
4. 根结点包含样本全集
5. 从根结点到每个叶结点的路径对应了一个判定测试序列

决策树学习的目的是为了产生一棵**泛化能力强**, 即处理未见示例能力强的决策树, 其基本流程遵循简单且直观的**"分而治之"** (divide-and-conquer)策略

4.1.3 决策树的基本算法流程

1 决策树的基本算法

输入: 训练集 $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;
 属性集 $A = \{a_1, a_2, \dots, a_d\}$.
 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 $A = \emptyset$ 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

图 4.2 决策树学习基本算法

2 具体递归过程

决策树的生成是一个递归过程, 在决策树基本算法中, 有三种情形会导致递归返回:

1. 当前结点包含的样本全属于同一类别, 无需划分;
2. 当前属性集为空, 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.

在第 (2) 种情形下, 把当前结点标记为叶结点, 并将其类别设定为该结点所含样本最多的类别;

在第 (3) 种情形下, 同样把当前结点标记为叶结点, 但将其类别设定为其父结点所含样本最多的类别.

3 递归过程的详细解析

1. 这是最理想的情况, 也即是子集只包含同一类别的样本. 若递归划分过程中某子集已经只含有某一类别的样本 (例如只含好瓜或坏瓜), 那么此时划分的目的已经达到了, 无需再进行划分, 这种情况就是**递归返回的情形 1**
2. 递归划分时每次选择一个属性, 并且**划分依据属性不能重复使用** (从候选依据属性中将当前使用的依据属性剔除, 这是因为根据某属性划分之后, 产生的各个子集在该属性上的取值相同). 但样本的属性是有限的, 因此划分次数不超过属性个数; 若所有属性均已被作为划分依据, 此时子集中仍含有多类样本 (例如仍然同时含有好瓜和坏瓜). 但是因已无属性可用作划分依据 (即子集中样本在各属性上取值均相同, 但却无法达到子集只包含同一类别的样本). 此时只能少数服从多数, 以此子集中样本数最多的类为标记, 此即为**递归返回的情形 2 中的 $A = \emptyset$**
3. 每递归一次, 候选的属性个数就减少一个. 假如现在还剩下两个属性, 触感和色泽. 且此时样本是多类的 (即好瓜和坏瓜). 但是剩下的样本触感和色泽都是一样的, 即当前样本集合在任何候选属性上的取值相同, 无法再通过属性划分了. (两个以上属性的情况类似). 此时也只能少数服从多数, 以此子集中样本数最多的类为标记, 此即为**递归返回情形 2 中的 " D 中样本在 A 上取值相同"**

4. 根据某属性进行划分时, 若该属性多个属性值中的某个属性值不包含任何样本. 如当前子集 D_v 以"色泽"属性来划分, "色泽"的属性值有: "青绿", "乌黑"和"浅白". 可发现没有样本的属性值为"浅白"的, 只有"青绿"和"乌黑". 此时对于取值为"浅白"的分支, 因为没有样本落入, 将其标记为叶结点, 其类别标记为 D_v 中样本最多的类.(注意, 此分支必须保留, 因为在测试时, 可能会有样本落入该分支). 此种情况即为**递归返回的情形 3**.

3.2 决策树的划分(ID3)

决策树学习的关键点在于: 如何选择最优划分属性.

希望决策树的分支结点所包含的样本尽可能属于同一类别, 即结点的"纯度"(purity)越来越高.

4.2.1 信息增益

1 信息熵 (香农熵)

"信息熵"(information entropy)是对量样本集合纯度最常用的一种指标. 它的定义如下:

假定当前样本集合 D 中第 k 类所占的比例为 $p_k (k = 1, 2, \dots, |\mathcal{Y}|)$, 则 D 的信息熵定义为:

$$\text{Ent}(D) = - \sum_{k=1}^{|\mathcal{Y}|} p_k \log_2 p_k \quad (3.1)$$

$\text{Ent}(D)$ 的值越小, 则 D 的纯度越高.

注: 这里的 k 就是西瓜里的"好瓜"和"坏瓜"这两个类别

2 信息增益

假定离散属性 a 有 V 个可能的取值 $\{a^1, a^2, \dots, a^V\}$, 若使用 a 来对样本集 D 进行划分, 则会产生 V 个分支结点, 其中第 v 个分支结点包含了 D 中所有在属性 a 上取值为 a^v 的样本, 记为 D^v . 可根据 (4.1) 计算出 D^v 的信息熵, 同时再考虑到不同分支结点所包含的样本数不同, 给分支结点赋予权重 $|D^v| / |D|$, 即样本数越多的分支结点的影响越大, 于是可计算出**用属性 a 对样本集 D 进行划分所获得的"信息增益"** (information gain)

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Ent}(D^v) \quad (3.2)$$

一般而言, **信息增益越大**, 则意味着使用属性 a 来进行划分所获得的**"纯度提升"越大**. 因此, 我们可以用信息增益来进行决策树的划分属性选择, 即在图4.2算法第 8 行选择属性 $a_* = \arg \max_{a \in A} \text{Gain}(D, a)$. ID3

决策树学习算法就是基于信息增益来进行属性划分的.

注: 信息增益准则对**可取值数目较多**的属性有**偏好**

注: 关于公式 3.2 中的各项理解

1. 条件熵: 条件熵表示在条件 X 下 Y 的信息熵. 公式如下:

$$H(Y|X) = \sum_{x \in X} p(x) H(Y|X = x)$$

也就是 3.2 中的 $\sum_{v=1}^V \frac{|D^v|}{|D|} \text{Ent}(D^v)$ 部分.

2. 信息增益 = 信息熵 - 条件熵

3.3 决策树的原理

3.3.1 决策树工作原理

如何构造一个决策树? 我们使用 createBranch() 方法, 如下所示:

```
def createBranch():  
    '''  
    此处运用了迭代的思想  
    '''  
    检测数据集中的所有数据的分类标签是否相同:  
    If so return 类标签  
    Else:  
        寻找划分数据集的最好特征 (划分之后信息熵最小, 也就是信息增益最大的特征)  
        划分数据集  
        创建分支节点  
        for 每个划分的子集  
            调用函数 createBranch (创建分支的函数) 并增加返回结果到分支节点中  
    return 分支节点
```

上面的伪代码createBranch是一个递归函数, 在倒数第二行直接调用了它自己.

3.3.2 决策树的一般流程

- 收集数据: 可以使用任何方法.
- 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.
- 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.
- 训练算法: 构造树的数据结构.
- 测试算法: 使用训练好的树计算错误率.
- 使用算法: 此步骤可以适用于任何监督学习任务, 而使用决策树可以更好地理解数据的内在含义.

3.3.3 决策树算法的特点

- 优点: 计算复杂度不高, 输出结果易于理解, 数据有缺失也能跑, 可以处理不相关特征.
- 缺点: 容易过拟合.
- 适用数据类型: 数值型和标称型.

3.4 决策树实战之判定鱼类

3.4.1 项目概述

根据以下 2 个特征, 将动物分成两类: 鱼类和非鱼类。

特征:

1. 不浮出水面是否可以生存
2. 是否有脚蹼

3.4.2 开发流程

- 收集数据: 可以使用任何方法.
- 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.
- 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.
- 训练算法: 构造树的数据结构.
- 测试算法: 使用训练好的树计算错误率.
- 使用算法: 此步骤可以适用于任何监督学习任务, 而使用决策树可以更好地理解数据的内在含义.

1 收集数据: 可以使用任何方法

在这里,我们构造 createDataSet() 函数输入书上例子的数据.如下图

表3-1 海洋生物数据

	不浮出水面是否可以生存	是否有脚蹼	属于鱼类
1	是	是	是
2	是	是	是
3	是	否	否
4	否	是	否
5	否	是	否

具体代码如下:

```
"""
函数说明：创建数据集

Parameters:
    无
Returns:
    dataSet - 数据集
    features - 特征(属性)
"""
def createDataSet():
    dataSet = [[1,1,'yes'],
               [1,1,'yes'],
               [1,0,'no'],
               [0,1,'no'],
               [0,1,'no']]
    features = ['不需要浮出水面', '脚蹼']
    return dataSet, features

if __name__ == '__main__':
    dataSet, features = createDataSet()
    print(dataSet)
```

```
print(features)
```

具体结果如下:

```
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
```

```
IPython 6.2.1 -- An enhanced Interactive Python.
```

```
In [1]: runfile('C:/Users/HeatonHsu/.spyder-py3/temp.py', wdir='C:/Users/HeatonHsu/.spyder-py3')
```

```
In [2]: runfile('C:/Users/HeatonHsu/.spyder-py3/temp.py', wdir='C:/Users/HeatonHsu/.spyder-py3')
[[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']] ['不需要浮出水面', '脚蹼']
```

```
In [3]:
```

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. 但我们输入的数据本身已经离散化了, 所以这一步就可以省略了.

3 分析数据 (核心部分)

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.

(1) 计算信息熵 (香农熵)

信息熵的计算公式详见公式 3.1

构造计算给定数据集的信息熵(香浓熵)的函数—calcShannonEnt()

具体代码如下:

```
from math import log

"""
函数说明: 计算熵

Parameters:
    无
Returns:
    dataSet - 数据集
    features - 特征(属性)
"""

def createDataSet():
    dataSet = [[1,1,'yes'],
               [1,1,'yes'],
               [1,0,'no'],
               [0,1,'no'],
               [0,1,'no']]
    features = ['不需要浮出水面', '脚蹼']
    return dataSet, features

"""
函数说明: 计算给定数据集的香农熵
```

```

Parameters:
    dataSet - 数据集
Returns:
    shannonEnt - 香农熵
"""
def calcShannonEnt(dataSet):
    #返回数据集的元素个数(dataSet是列表,且每个元素也是一个列表)
    numEntires = len(dataSet)
    #设置一个空字典,用于保存每个标签(label,即类别)和标签出现的次数
    labelCounts = {}
    #循环提取每个样本(也就是dataSet的每个元素)的标签(label)
    for featVec in dataSet:
        #提取当前的标签
        currentLabel = featVec[-1]
        #如果当前的标签不在字典labelCount的key中,那么就添加进去
        #labelCount.keys()返回的是dic_keys对象,类似于列表等对象,可进行迭代,也可进行逻辑运算
        if currentLabel not in labelCounts.keys():
            #且让currentLabel键对应的键值为0
            labelCounts[currentLabel] = 0
        #无论当前键出现或者未出现,对应的键值都加1,即可实现计数
        labelCounts[currentLabel] += 1
    #香农熵
    shannonEnt = 0
    #下面的是遍历字典中的所有键,它等价于labelCount.keys()
    #遍历字典中的所有值时用, labelCount.values()
    #遍历字典中的所有键值对时用, labelCount.items(),且需要两个变量来接收键和键值
    for key in labelCounts:
        #计算p(i)
        prob = float(labelCounts[key]) / numEntires
        #计算香农熵, shannonEnt起始为0,每次减去prob * log(prob, 2),等价于公式求和
        shannonEnt -= prob * log(prob, 2)
    return shannonEnt

if __name__ == '__main__':
    dataSet, features = createDataSet()
    shannoEnt = calcShannonEnt(dataSet)
    print(shannoEnt)

```

运行结果为:

```

Console 1/A ✕

In [8]: runfile('C:/Users/HeatonHsu/.spyder-py3/temp.py', wdir='C:/Users/HeatonHsu/.spyder-py3')
0.9709505944546686

In [9]:

```

(2) 按照给定特征划分数据集

为计算条件熵, 那么就需要知道在给定特征划分后的数据集.

添加函数 splitDataSet, 具体代码如下:

```

"""
函数说明: 计算熵

Parameters:
    无

```

```

Returns:
    dataSet - 数据集
    features - 特征(属性)
"""
def createDataSet():
    dataSet = [[1,1,'yes'],
               [1,1,'yes'],
               [1,0,'no'],
               [0,1,'no'],
               [0,1,'no']]
    features = ['不需要浮出水面', '脚蹼']
    return dataSet, features

"""
函数说明：计算给定数据集的香农熵
"""

函数说明：按照给定的特征划分数据集

splitDataSet(通过遍历dataSet数据集， 求出index对应的colNum列的值为value的行)
就是依据index列进行分类， 如果index列的数据等于 value的时候， 就要将 index 划分到我们创建的
新的数据集中

Parameters:
    dataSet - 待划分的数据集
    axis - 划分数据集的特征(在这个例子里共两个),即每一行的axis列
    value - 需要返回的该特征的值
Returns:
    retDataSet - 划分后的数据集
"""
def splitDataSet(dataSet, axis, value):
    #创建列表,用于接收返回的数据集列表
    retDataSet = []
    #遍历数据集(遍历每一个样本)
    for featVec in dataSet:
        #如果该特征的值也就是featVec[axis]等于value
        if featVec[axis] == value:
            #那么先将featVec中axis元素之前的元素存到reducedFeatVec中
            reducedFeatVec = featVec[:axis]
            #接着再把axis之后的元素添加到reduceFeatVec中。
            #这样就把featVec[axis]这个元素剔除了
            reducedFeatVec.extend(featVec[axis+1:])
            #最后把处理好的reducedFeatVec都存储再retDataSet中
            retDataSet.append(reducedFeatVec)
    return retDataSet

if __name__ == '__main__':
    dataSet,features = createDataSet()
    retDataSet = splitDataSet(dataSet,0,1)    #用axis=0,value=1来划分数据集
    print(retDataSet)

```

用axis=0,value=1(即第一个特征的值为1)来划分数据集的结果如下:

```

In [10]: runfile('C:/Users/HeatonHsu/.spyder-py3/temp.py', wdir='C:/Users/HeatonHsu/.spyder-py3')
[[1, 'yes'], [1, 'yes'], [0, 'no']]

```

```

In [11]:

```


本段代码需要注意的点有:

1. 关于append和extend的区别

具体差别可参看下面的运行结果

```
In [18]: a = [1, 2, 3]
In [19]: b = [1, 2, 3, 4]
In [20]: c = [7, 8, 9]
In [21]: a.extend(b)
In [22]: a
Out[22]: [1, 2, 3, 1, 2, 3, 4]
In [23]: b.append(c)
In [24]: b
Out[24]: [1, 2, 3, 4, [7, 8, 9]]
```

(3) 选择最优的数据集划分方式 (即信息增益最大)

添加chooseBestFeatureToSplit函数, 具体代码如下:

```
from math import log

"""
函数说明: 计算熵

Parameters:
    无
Returns:
    dataSet - 数据集
    features - 特征(属性)
"""
def createDataSet():
    dataSet = [[1, 1, 'yes'],
               [1, 1, 'yes'],
               [1, 0, 'no'],
               [0, 1, 'no'],
               [0, 1, 'no']]
    features = ['不需要浮出水面', '脚蹼']
    return dataSet, features

"""
函数说明: 计算给定数据集的香农熵

Parameters:
    dataSet - 数据集
Returns:
    shannonEnt - 香农熵
"""
def calcShannonEnt(dataSet):
    #返回数据集的元素个数(dataSet是列表,且每个元素也是一个列表)
```

```

numEntires = len(dataSet)
#设置一个空字典,用于保存每个标签(label,即类别)和标签出现的次数
labelCounts = {}
#循环提取每个样本(也就是dataSet的每个元素)的标签(label)
for featVec in dataSet:
    #提取当前的标签
    currentLabel = featVec[-1]
    #如果当前的标签不在字典labelCount的key中,那么就添加进去
    #labelCount.keys()返回的是dic_keys对象,类似于列表等对象,可进行迭代,也可进行逻辑运算
    if currentLabel not in labelCounts.keys():
        #且让currentLabel键对应的键值为0
        labelCounts[currentLabel] = 0
    #无论当前键出现或者未出现,对应的键值都加1,即可实现计数
    labelCounts[currentLabel] += 1
#香农熵
shannonEnt = 0
#下面的是遍历字典中的所有键,它等价于labelCount.keys()
#遍历字典中的所有值时用, labelCount.values()
#遍历字典中的所有键值对时用, labelCount.items(),且需要两个变量来接收键和键值
for key in labelCounts:
    #计算p(i)
    prob = float(labelCounts[key]) / numEntires
    #计算香农熵, shannonEnt起始为0,每次减去prob * log(prob, 2),等价于公式求和
    shannonEnt -= prob * log(prob, 2)
return shannonEnt

```

"""

函数说明: 按照给定的特征划分数据集

splitDataSet(通过遍历dataSet数据集, 求出index对应的colnum列的值为value的行)
就是依据index列进行分类, 如果index列的数据等于 value的时候, 就要将 index 划分到我们创建的新的数据集中

Parameters:

dataSet - 带划分的数据集
axis - 划分数据集的特征(在这个例子里共两个)
value - 需要返回的该特征的值

Returns:

retDataSet - 划分后的数据集

"""

```

def splitDataSet(dataSet, axis, value):
    #创建列表,用于接收返回的数据集列表
    retDataSet = []
    #遍历数据集(遍历每一个样本)
    for featVec in dataSet:
        #如果该特征的值也就是featVec[axis]等于value
        if featVec[axis] == value:
            #那么先将featVec中axis元素之前的元素存到reducedFeatVec中
            reducedFeatVec = featVec[:axis]
            #接着再把axis之后的元素添加到reduceFeatVec中.
            #这样就把featVec[axis]这个元素剔除了
            reducedFeatVec.extend(featVec[axis+1:])
            #最后把处理好的reducedFeatVec都存储再retDataSet中
            retDataSet.append(reducedFeatVec)
    return retDataSet

```

"""

函数说明：计算信息增益,选择最优的特征

Parameters:

 dataset - 数据集

Returns:

 bestFeature - 信息增益最大的特征的索引值

"""

```
def chooseBestFeatureToSplit(dataset):
    #计算出特征的数量,最后一个元素是label(类别),因此减一
    numFeatures = len(dataset[0]) - 1
    #计算出原始数据集的香农熵
    baseEntropy = calcShannonEnt(dataset)
    #初始化最优的信息增益
    bestInforGain = 0
    #初始化最优特征的编号
    bestFeature = -1
    #遍历所有的特征
    for i in range(numFeatures):
        #获取dataset的第i个特征的所有值(即每个样本的第i个元素)
        #下面的用的是列表生成式,相关知识查阅python基础语法
        featList = [example[i] for example in dataset]
        #创建uniqueVals集合(集合中元素唯一,不重合)
        uniqueVals = set(featList)
        #经验条件熵
        newEntropy = 0
        #计算信息增益
        #遍历某一列的value集合, 计算该列的信息熵
        #遍历当前特征(i)中的所有唯一属性值,对每个唯一属性值划分一次数据集,计算当前数据集的新熵
        #值,并熵求和。
        for value in uniqueVals:
            #subDataSet表示划分后的子集合
            subDataSet = splitDataSet(dataset,i,value)
            #计算当前子集的概率
            prob = len(subDataSet) / float(len(dataset))
            #计算条件熵
            newEntropy += prob * calcShannonEnt(subDataSet)
        #计算当前特征(i)的信息增益
        inforGain = baseEntropy - newEntropy
        #打印每个特征的信息增益
        print("第%d个特征的信息增益为%.2f" % (i, inforGain))
        #通过比较找出最大的增益
        if inforGain > bestInforGain:
            bestInforGain = inforGain
            bestFeature = i
    return bestFeature

if __name__ == '__main__':
    dataset,features = createDataSet()
    print("根据信息增益,最优特征的索引为: " + str(chooseBestFeatureToSplit(dataset)))
```

运行结果为:

```
In [12]: runfile('C:/Users/HeatonHsu/.spyder-py3/temp.py', wdir='C:/Users/HeatonHsu/.spyder-py3')
第0个特征的信息增益为0.42
第1个特征的信息增益为0.17
根据信息增益,最优特征的索引为: 0
```

```
In [13]:
```

4 训练算法 (核心部分)

训练算法: 构造树的数据结构

ID3算法的核心是在决策树各个结点上对应信息增益准则选择特征, 递归地构建决策树. 具体方法是: 从根结点(root node)开始, 对结点计算所有可能的特征的信息增益, 选择信息增益最大的特征作为结点的特征, 由该特征的不同取值建立子节点; 再对子结点递归地调用以上方法, 构建决策树; 直到所有特征的信息增益均很小或没有特征可以选择为止. 最后得到一个决策树. ID3相当于用极大似然法进行概率模型的选择.

在先构造树前, 先要知道两者结束的条件:

1. 是最后的样本的类别都相同
2. 还有就是所有属性都划分完, 这时用类别出现最多的作为这个子集的类别.

对于第一个很好处理, 第二个我们需要构造一个 **majorityCnt 函数**用于计算当前子集的类别.

具体代码如下:

```
"""
函数说明: 统计classList中出现最多的元素(即标签), 类似于前面的classify0

Parameters:
    classList - 标签列表
Returns:
    sortedClassCount[0][0] - 出现次数最多的元素(标签)
"""
def majorityCnt(classList):
    #可以参考前面的classify0函数, 因此下面的代码可以改写为:
    """
    classCount = {}
    for vote_1 in classList:
        classCount[vote_1] = classCount.get(vote_1, 0) + 1
    sortedClassCount_3 =
sorted(classCount.items(), key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]
    """
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
        classCount[vote] += 1
    sortedClassCount =
sorted(classCount.items(), key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]

if __name__ == '__main__':
    classList = ['yes', 'no', 'yes', 'yes', 'no']
    result = majorityCnt(classList)
    print(result)
```

接着构造树, 创建creatTree函数, 具体代码如下:

```
"""
```

函数说明:创建决策树

Parameters:

dataSet - 数据集

features - 特征(属性)(按顺序)

"""

```
def createTree(dataSet, features):
```

```
    #获取各个样本的分类标签
```

```
    classList = [example[-1] for example in dataSet]
```

```
    #如果类别全部相同,那么停止划,这是第一种情况
```

```
    #如果数据集的最后一列的第一个值出现的次数=整个集合的数量, 也就说只有一个类别, 就只直接返回结果就行
```

```
    #第一个停止条件: 所有的类标签完全相同, 则直接返回该类标签。
```

```
    if classList.count(classList[0]) == len(classList):
```

```
        return classList[0]
```

```
    #如果数据集只有1列(也就是只有类别标签一列了),那么选出出现label次数最多的一类,作为结果
```

```
    #第二个停止条件:使用完了所有特征,仍然不能将数据集划分成仅包含唯一类别的分组
```

```
    if len(dataSet[0]) == 1:
```

```
        return majorityCnt(classList)
```

```
    #选择最优的特征的索引
```

```
    bestFeat = chooseBestFeatureToSplit(dataSet)
```

```
    #根据bestFeat的索引获取最优特征的名称
```

```
    bestFeatLabel = features[bestFeat]
```

```
    #初始化myTree
```

```
    myTree = {bestFeatLabel:{}}
```

```
    #删除已使用的特征
```

```
    del(features[bestFeat])
```

```
    #获取集合中所有最优特征的值
```

```
    featValues = [example[bestFeat] for example in dataSet]
```

```
    #去掉重复的属性值
```

```
    uniqueVals = set(featValues)
```

```
    #遍历特征,创建决策树
```

```
    for value in uniqueVals:
```

```
        #剩余的标签
```

```
        subLabels = features[:]
```

```
        # 遍历当前选择特征包含的所有属性值, 在每个数据集划分上递归调用函数createTree()
```

```
        myTree[bestFeatLabel][value] =
```

```
createTree(splitDataSet(dataSet,bestFeat,value),subLabels)
```

```
    return myTree
```

注: 最后的递归我也不是很熟,后续再看看相关材料.其他的都还好.

最后,把前面的几个函数放在一起,组成完整的构造决策树的代码块,如下:

```
from math import log
```

```
"""
```

函数说明: 计算熵

Parameters:

无

Returns:

dataSet - 数据集

features - 特征(属性)

```
"""
```

```
def createDataSet():
    dataSet = [[1,1,'yes'],
               [1,1,'yes'],
               [1,0,'no'],
               [0,1,'no'],
               [0,1,'no']]
    features = ['不需要浮出水面', '脚蹼']
    return dataSet, features
```

```
"""
```

函数说明：计算给定数据集的香农熵

Parameters:

dataSet - 数据集

Returns:

shannonEnt - 香农熵

```
"""
```

```
def calcShannonEnt(dataSet):
    #返回数据集的元素个数(dataSet是列表,且每个元素也是一个列表)
    numEntires = len(dataSet)
    #设置一个空字典,用于保存每个标签(label,即类别)和标签出现的次数
    labelCounts = {}
    #循环提取每个样本(也就是dataSet的每个元素)的标签(label)
    for featVec in dataSet:
        #提取当前的标签
        currentLabel = featVec[-1]
        #如果当前的标签不在字典labelCount的key中,那么就添加进去
        #labelCount.keys()返回的是dic_keys对象,类似于列表等对象,可进行迭代,也可进行逻辑运算
        if currentLabel not in labelCounts.keys():
            #且让currentLabel键对应的键值为0
            labelCounts[currentLabel] = 0
        #无论当前键出现或者未出现,对应的键值都加1,即可实现计数
        labelCounts[currentLabel] += 1
    #香农熵
    shannonEnt = 0
    #下面的是遍历字典中的所有键,它等价于labelCount.keys()
    #遍历字典中的所有值时用, labelCount.values()
    #遍历字典中的所有键值对时用, labelCount.items(),且需要两个变量来接收键和键值
    for key in labelCounts:
        #计算p(i)
        prob = float(labelCounts[key]) / numEntires
        #计算香农熵, shannonEnt起始为0,每次减去prob * log(prob, 2),等价于公式求和
        shannonEnt -= prob * log(prob, 2)
    return shannonEnt
```

```
"""
```

函数说明：按照给定的特征划分数据集

splitDataSet(通过遍历dataSet数据集， 求出index对应的colnum列的值为value的行)

就是依据index列进行分类， 如果index列的数据等于 value的时候， 就要将 index 划分到我们创建的新的数据集中

Parameters:

dataSet - 带划分的数据集

axis - 划分数据集的特征(在这个例子里共两个)

value - 需要返回的该特征的值

Returns:

retDataSet - 划分后的数据集

"""

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

```
#创建列表,用于接收返回的数据集列表
```

```
retDataSet = []
```

```
#遍历数据集(遍历每一个样本)
```

```
for featVec in dataset:
```

```
#如果该特征的值也就是featVec[axis]等于value
```

```
if featVec[axis] == value:
```

```
#那么先将featVec中axis元素之前的元素存到reducedFeatVec中
```

```
reducedFeatVec = featVec[:axis]
```

```
#接着再把axis之后的元素添加到reduceFeatVec中.
```

```
#这样就把featVec[axis]这个元素剔除了
```

```
reducedFeatVec.extend(featVec[axis+1:])
```

```
#最后把处理好的reducedFeatVec都存储再retDataSet中
```

```
retDataSet.append(reducedFeatVec)
```

```
return retDataSet
```

"""

函数说明: 计算信息增益,选择最优的特征

Parameters:

dataset - 数据集

Returns:

bestFeature - 信息增益最大的特征的索引值

"""

```
def chooseBestFeatureToSplit(dataset):
```

```
#计算出特征的数量,最后一个元素是label(类别),因此减一
```

```
numFeatures = len(dataset[0]) - 1
```

```
#计算出原始数据集的香农熵
```

```
baseEntropy = calcShannonEnt(dataset)
```

```
#初始化最优的信息增益
```

```
bestInforGain = 0
```

```
#初始化最优特征的编号
```

```
bestFeature = -1
```

```
#遍历所有的特征
```

```
for i in range(numFeatures):
```

```
#获取dataset的第i个特征的所有值(即每个样本的第i个元素)
```

```
#下面的用的是列表生成式,相关知识查阅python基础语法
```

```
featList = [example[i] for example in dataset]
```

```
#创建uniqueVals集合(集合中元素唯一,不重合)
```

```
uniqueVals = set(featList)
```

```
#经验条件熵
```

```
newEntropy = 0
```

```
#计算信息增益
```

```
#遍历某一列的value集合, 计算该列的信息熵
```

```
#遍历当前特征(i)中的所有唯一属性值,对每个唯一属性值划分一次数据集,计算当前数据集的新熵值,并熵求和。
```

```
for value in uniqueVals:
```

```
#subDataSet表示划分后的子集合
```

```
subDataSet = splitDataSet(dataset,i,value)
```

```
#计算当前子集的概率
```

```
prob = len(subDataSet) /float(len(dataset))
```

```
#计算条件熵
```

```
newEntropy += prob * calcShannonEnt(subDataSet)
```

```
#计算当前特征(i)的信息增益
```

```
inforGain = baseEntropy - newEntropy
```

```

        #打印每个特征的信息增益
        #print("第%d个特征的信息增益为%.2f" % (i, inforGain))
        #通过比较找出最大的增益
        if inforGain > bestInforGain:
            bestInforGain = inforGain
            bestFeature = i
    return bestFeature

"""
函数说明：统计classList中出现最多的元素(即标签),类似于前面的classify0

Parameters:
    classList - 标签列表
Returns:
    sortedClassCount[0][0] - 出现次数最多的元素(标签)
"""
def majorityCnt(classList):
    #可以参考前面的classify0函数,因此下面的代码可以改写为:
    """
    classCount = {}
    for vote_1 in classList:
        classCount[vote_1] = classCount.get(vote_1,0) + 1
    sortedClassCount_3 =
sorted(classCount.items(),key=operator.itemgetter(1),reverse=True)
    """
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
            classCount[vote] += 1
    sortedClassCount =
sorted(classCount.items(),key=operator.itemgetter(1),reverse=True)
    return sortedClassCount[0][0]

"""
函数说明：创建决策树

Parameters:
    dataSet - 数据集
    features - 特征(属性)
    featLabels - 用于存储选择的最优属性(特征)标签
"""
def createTree(dataSet, features):
    #获取各个样本的分类标签
    classList = [example[-1] for example in dataSet]
    #如果类别全部相同,那么停止划分
    # 如果数据集的最后一列的第一个值出现的次数=整个集合的数量, 也就说只有一个类别, 就只直接
    返回结果就行
    # 第一个停止条件：所有的类标签完全相同, 则直接返回该类标签。
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    # 如果数据集只有1列(也就是只有类别标签一列了), 那么选出出现label次数最多的一类, 作为结
    果
    # 第二个停止条件：使用完了所有特征, 仍然不能将数据集划分成仅包含唯一类别的分组
    if len(dataSet[0]) == 1:
        return majorityCnt(classList)

```



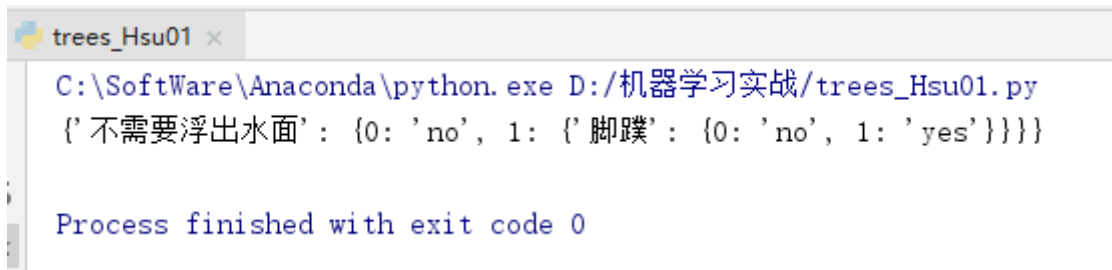
```

#选择最优的特征
bestFeat = chooseBestFeatureToSplit(dataSet)
#根据bestFeat的值(数字)获取最优特征的名称
bestFeatLabel = features[bestFeat]
#初始化myTree
myTree = {bestFeatLabel: {}}
#删除已使用的特征
del(features[bestFeat])
#获取集合中所有最优特征的值
featValues = [example[bestFeat] for example in dataSet]
#去掉重复的属性值
uniqueVals = set(featValues)
#遍历特征,创建决策树
for value in uniqueVals:
    #剩余的标签
    subLabels = features[:]
    # 遍历当前选择特征包含的所有属性值, 在每个数据集划分上递归调用函数createTree()
    myTree[bestFeatLabel][value] =
createTree(splitDataSet(dataSet,bestFeat,value),subLabels)
return myTree

if __name__ == '__main__':
    dataSet, features = createDataSet()
    featLabels = ['不需要浮出水面', '脚蹼']
    myTree = createTree(dataSet,features)

```

运行结果如下(打印各个特征信息增益的注释了,不显示):



```

trees_Hsu01 x
C:\SoftWare\Anaconda\python.exe D:/机器学习实战/trees_Hsu01.py
{'不需要浮出水面': {0: 'no', 1: {'脚蹼': {0: 'no', 1: 'yes'}}}}

Process finished with exit code 0

```

5 测试算法(没涉及)

6 使用算法

使用算法: 使用生成好的决策树执行分类,

创建classify分类函数, 具体代码如下:

```

"""
函数说明:使用决策树执行分类

Parameters:
    inputTree - 已经生成的决策树
    featLabels - Feature(特征)标签对应的名称
    testVec - 测试数据列表
Returns:

```

```

classLabel - 分类结果
"""

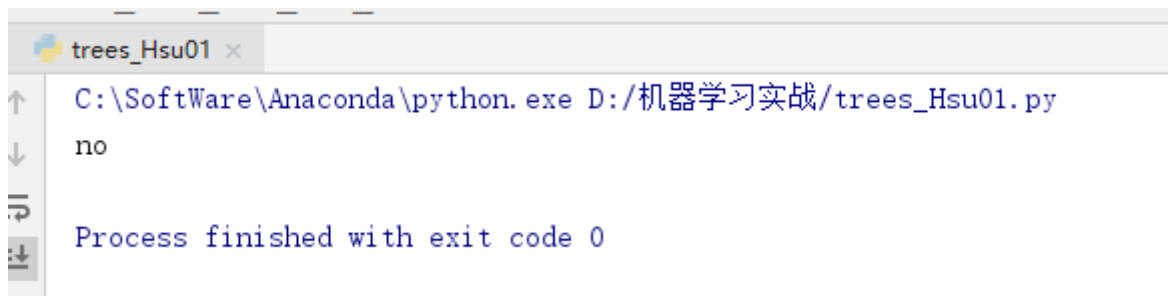
def classify(inputTree, featLabels, testVec):
    #获取inputTree的根节点key(键名)
    firstStr = list(inputTree.keys())[0]
    #通过key得到根节点对应的value
    secondDict = inputTree[firstStr]
    #获取featLabels里firstStr的索引,即判断根节点名称获取根节点在label中的先后顺序
    featIndex = featLabels.index(firstStr)
    for key in secondDict.keys():
        if testVec[featIndex] == key:
            #如果里面还是字典的话,说明还需要进一步进行上述操作,即进行递归
            if isinstance(secondDict[key],dict):
                classLabel = classify(secondDict[key],featLabels,testVec)
            #否则, 就是已经分类完了,那么secondDict[key],即结果no或者yes
            else:
                classLabel = secondDict[key]
    return classLabel

if __name__ == '__main__':
    dataSet, features = createDataSet()
    featLabels = ['不需要浮出水面', '脚蹼']
    myTree = createTree(dataSet,features)
    testVec = [0,1]
    result = classify(myTree, featLabels,testVec)
    print(result)

```

和上面的构造决策树的代码组合在一起, 构成完整的测试(合在一起太长了,这里就不展示了)

结果如下:



```

trees_Hsu01 x
C:\SoftWare\Anaconda\python.exe D:/机器学习实战/trees_Hsu01.py
no
Process finished with exit code 0

```

结果是非鱼类.

3.5 决策树的存储和读取

如果我们通过数据构造好了决策树, 那么如何更好的存储它和读取使用它.

构造决策树是很耗时的任务, 即使处理很小的数据集, 如前面的样本数据, 也要花费几秒的时间, 如果数据集很大, 将会耗费很多计算时间. 然而用创建好的决策树解决分类问题, 则可以很快完成. 因此, 为了节省计算时间, 最好能够在每次执行分类时调用已经构造好的决策树. 为了解决这个问题, 需要使用Python模块pickle序列化对象. 序列化对象可以在磁盘上保存对象, 并在需要的时候读取出来.

存储决策树:

具体代码如下:

```
import pickle

"""
函数说明: 存储决策树

Parameters:
    inputTree - 已经生成的决策树
    filename - 存储决策树的文件名
Returns:
    无
"""
def storeTree(inputTree, filename):
    #以写入形式打开文件,用于后续写入文件
    with open(filename, 'wb') as f:
        #pickle序列化对象并存储起来
        pickle.dump(inputTree, f)

if __name__ == '__main__':
    myTree = {'不需要浮出水面': {0: 'no', 1: {'脚蹼': {0: 'no', 1: 'yes'}}}}
    storeTree(myTree, '树结构.txt')
```

那么我们的树就存储在当前文件夹下的"树结构.txt"这个文件中

接着读取树, 具体代码如下:

```
"""
函数说明: 读取决策树

Parameters:
    filename - 决策树的存储文件名
Returns:
    myTree - 读取出来的决策树(字典形式)
"""
def grabTree(filename):
    f = open(filename, 'rb')
    return pickle.load(f)

if __name__ == '__main__':
    myTree = grabTree('树结构.txt')
    print(myTree)
```

小结:

1. 关于递归, 后续还需继续查阅学习
2. 关于决策树的可视化, 这篇没有涉及, 树上提供的是自己一步步手写实现的, 关于这部分, 我看了一遍, 太繁琐了. 其实是有相关的模块可以实现, 以后再说吧

下一节继续介绍如何使用sklearn模块来实现决策树 (有轮子当然用轮子)

