决策树

决策树是一种机器学习算法，用于回归和分类任务。它通过**对数据集进行递归划分**，构建一个树状结构，**每个节点代表一个特征属性，每个分支代表一个特征值，而每个叶节点代表一个类别标签或者回归值。**决策树的构建过程基于对特征的选择，目标是在每个节点处选择最能有效划分数据的特征。常见的划分策略有信息增益、基尼不纯度等，这些方法衡量了每个特征在划分后对数据集纯度提升的程度。

在分类问题中，表示基于特征对实例进行分类的过程，可以认为是if-then的集合，也可以认为是定义在特征空间与类空间上的条件概率分布。

决策树通常有三个步骤：**特征选择、决策树的生成、决策树的修剪。**

用决策树分类：从根节点开始，对实例的某一特征进行测试，根据测试结果将实例分配到其子节点，此时每个子节点对应着该特征的一个取值，如此递归的对实例进行测试并分配，直到到达叶节点，最后将实例分到叶节点的类中。

下图为决策树示意图，圆点——内部节点，方框——叶节点

决策树学习的目标：根据给定的训练数据集构建一个决策树模型，使它能够对实例进行正确的分类。

决策树学习的本质：从训练集中归纳出一组分类规则，或者说是由训练数据集估计条件概率模型。

决策树学习的损失函数：正则化的极大似然函数

决策树学习的测试：最小化损失函数

决策树学习的目标：在损失函数的意义下，选择最优决策树的问题。

决策树原理和问答猜测结果游戏相似，根据一系列数据，然后给出游戏的答案。

k-近邻算法可以完成很多分类任务，但是其最大的缺点是无法给出数据的内在含义，决策树的优势在于数据形式非常容易理解。

下面是决策树的基本构建步骤：

1. **选择特征：构建根节点，将所有训练数据都放在根节点，选择一个最优特征，按着这一特征将训练数据集分割成子集，使得各个子集有一个在当前条件下最好的分类。**
2. 如果这些子集已经能够被基本正确分类，那么构建叶节点，并将这些子集分到所对应的叶节点去。
3. **递归构建：如果还有子集不能够被正确的分类，那么就对这些子集选择新的最优特征，继续对其进行分割，构建相应的节点，如果递归进行，直至所有训练数据子集被基本正确的分类，或者没有合适的特征为止**
4. 每个子集都被分到叶节点上，即都有了明确的类，这样就生成了一颗决策树。

- 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的问题
- 适用数据类型：数值型和标称型

使用决策树做预测需要以下过程：

收集数据：可以使用任何方法。比如想构建一个相亲系统，我们可以从媒婆那里，或者通过参访相亲对象获取数据。根据他们考虑的因素和最终的选择结果，就可以得到一些供我们利用的数据了。
准备数据：收集完的数据，我们要进行整理，将这些所有收集的信息按照一定规则整理出来，并排版，方便我们进行后续处理。
分析数据：可以使用任何方法，决策树构造完成之后，我们可以检查决策树图形是否符合预期。
训练算法：这个过程也就是构造决策树，同样也可以说是决策树学习，就是构造一个决策树的数据结构。
测试算法：使用经验树计算错误率。当错误率达到了可接收范围，这个决策树就可以投放使用了。
使用算法：此步骤可以使用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。

划分数据集的大原则是：**将无序数据变得更加有序**，但是各种方法都有各自的优缺点，**信息论是量化处理信息的分支科学，在划分数据集前后信息发生的变化称为信息增益**，获得信息增益最高的特征就是最好的选择，所以必须先学习如何计算信息增益，**集合信息的度量方式称为香农熵**，或者简称熵。

希望通过所给的训练数据学习一个贷款申请的决策树，用以对未来的贷款申请进行分类，即当新的客户提出贷款申请时，根据申请人的特征利用决策树决定是否批准贷款申请。

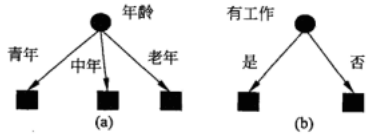

图 5.3　不同特征决定的不同决策树

特征选择就是决定用哪个特征来划分特征空间。比如，我们通过上述数据表得到两个可能的决策树，分别由两个不同特征的根结点构成。

图 (a) 所示的根结点的特征是年龄，有3个取值，对应于不同的取值有不同的子结点。

图 (b) 所示的根节点的特征是工作，有2个取值，对应于不同的取值有不同的子结点。两个决策树都可以从此延续下去。

问题是：究竟选择哪个特征更好些？这就要求确定选择特征的准则。

直观上，如果一个特征具有更好的分类能力，或者说，按照这一特征将训练数据集分割成子集，使得各个子集在当前条件下有最好的分类，那么就更应该选择这个特征。

信息增益就能够很好地表示这一直观的准则。

什么是信息增益呢？在划分数据集之前之后信息发生的变化成为信息增益，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。

熵定义为信息的期望值，如果待分类的事物可能划分在多个类之中，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所包含的信息期望值，通过下式得到：

$$H = -\Sigma_{i=1}^{n} p(x_i) log_2 p(x_i)$$

其中，n为分类数目，熵越大，随机变量的不确定性就越大。

当熵中的概率由数据估计(特别是最大似然估计)得到时，所对应的熵称为经验熵(empirical entropy)。

什么叫由数据估计？比如有10个数据，一共有两个类别，A类和B类。其中有7个数据属于A类，则该A类的概率即为十分之七。

其中有3个数据属于B类，则该B类的概率即为十分之三。浅显的解释就是，这概率是我们根据数据数出来的。

我们定义贷款申请样本数据表中的数据为训练数据集D，则训练数据集D的经验熵为H(D)，|D|表示其样本容量，及样本个数。

设有K个类Ck，k = 1,2,3,⋯,K，|Ck|为属于类Ck的样本个数，这经验熵公式可以写为：

$$H(D) = -\Sigma \frac{|c_k|}{|D|} log_2 \frac{|c_k|}{|D|}$$

根据此公式计算经验熵H(D)，分析贷款申请样本数据表中的数据。最终分类结果只有两类，即放贷和不放贷。根据表中的数据统计可知，在15个数据中，9个数据的结果为放贷，6个数据的结果为不放贷。所以数据集D的经验熵H(D)为：

$$H(D) = -\frac{9}{15} log_2 \frac{9}{15} - \frac{6}{15} log_2 \frac{6}{15} = 0.971$$

经过计算可知，数据集D的经验熵H(D)的值为0.971。

在理解信息增益之前，要明确——条件熵

信息增益表示得知特征 X 的信息而使得类Y的信息不确定性减少的程度。

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 (conditional entropy) H(Y|X)，定义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，$p_i = P(X = x_i)$

当熵和条件熵中的概率由数据估计（特别是极大似然估计）得到时，所对应的分别为经验熵和经验条件熵，此时如果有0概率，令 $0log0 = 0$

信息增益：信息增益是相对于特征而言的。所以，特征A对训练数据集D的信息增益g(D,A)，定义为集合D的经验熵H(D)与特征A给定条件下D的经验条件熵H(D|A)之差，即：

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

一般地，熵H(D)与条件熵H(D|A)之差成为互信息(mutual information)。决策树学习中的信息增益等价于训练数据集中类与特征的互信息。

信息增益值的大小相对于训练数据集而言的，并没有绝对意义，在分类问题困难时，也就是说在训练数据集经验熵大的时候，信息增益值会偏大，反之信息增益值会偏小，使用信息增益比可以对这个问题进行校正，这是特征选择的另一个标准。

信息增益比：特征A对训练数据集D的信息增益比$g_R(D, A)$定义为其信息增益$g(D, A)$与训练数据集D的经验熵之比：

$$g_R(D, A) = \frac{g(D, A)}{H(D)}$$

3.1.2 编写代码计算经验熵

在编写代码之前，我们先对数据集进行属性标注。

表 5.1　贷款申请样本数据表

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

年龄：0代表青年，1代表中年，2代表老年；
有工作：0代表否，1代表是；
有自己的房子：0代表否，1代表是；

信贷情况：0代表一般，1代表好，2代表非常好；
类别(是否给贷款)：no代表否，yes代表是。
创建数据集，计算经验熵的代码如下：

```
from math import log

"""
函数说明：创建测试数据集
Parameters：无
Returns：
    dataSet：数据集
    labels：分类属性
Modify：
    2018-03-12

"""
def creatDataSet():
    # 数据集
    dataSet=[[0, 0, 0, 0, 'no'],
            [0, 0, 0, 1, 'no'],
            [0, 1, 0, 1, 'yes'],
            [0, 1, 1, 0, 'yes'],
            [0, 0, 0, 0, 'no'],
            [1, 0, 0, 0, 'no'],
            [1, 0, 0, 1, 'no'],
            [1, 1, 1, 1, 'yes'],
            [1, 0, 1, 2, 'yes'],
            [1, 0, 1, 2, 'yes'],
            [2, 0, 1, 2, 'yes'],
            [2, 0, 1, 1, 'yes'],
            [2, 1, 0, 1, 'yes'],
            [2, 1, 0, 2, 'yes'],
            [2, 0, 0, 0, 'no']]
    #分类属性
    labels=['年龄','有工作','有自己的房子','信贷情况']
    #返回数据集和分类属性
    return dataSet,labels

"""
函数说明：计算给定数据集的经验熵（香农熵）
Parameters：
    dataSet：数据集
Returns：
    shannonEnt：经验熵
Modify：
    2018-03-12

"""
def calcShannonEnt(dataSet):
    #返回数据集行数
    numEntries=len(dataSet)
    #保存每个标签（label）出现次数的字典
    labelCounts={}
    #对每组特征向量进行统计
    for featVec in dataSet:
        currentLabel=featVec[-1]                     #提取标签信息
        if currentLabel not in labelCounts.keys():   #如果标签没有放入统计次数的字典，添加进去
            labelCounts[currentLabel]=0
        labelCounts[currentLabel]+=1                 #label计数

    shannonEnt=0.0                                   #经验熵
    #计算经验熵
    for key in labelCounts:
        prob=float(labelCounts[key])/numEntries      #选择该标签的概率
        shannonEnt-=prob*log(prob,2)                 #利用公式计算
    return shannonEnt                                #返回经验熵

#main函数
if __name__=='__main__':
    dataSet,features=creatDataSet()
    print(dataSet)
    print(calcShannonEnt(dataSet))
```

得到的结果是：

```
第0个特征的增益为0.083
第1个特征的增益为0.324
第2个特征的增益为0.420
第3个特征的增益为0.363
第0个特征的增益为0.252
第1个特征的增益为0.918
第2个特征的增益为0.474
{'有自己的房子': {0: {'有工作': {0: 'no', 1: 'yes'}}, 1: 'yes'}}
```

利用代码计算信息增益

```
from math import log

"""
函数说明：创建测试数据集
Parameters：无
Returns：
    dataSet：数据集
    labels：分类属性
Modify：
    2018-03-12

"""
def creatDataSet():
    # 数据集
    dataSet=[[0, 0, 0, 0, 'no'],
            [0, 0, 0, 1, 'no'],
            [0, 1, 0, 1, 'yes'],
            [0, 1, 1, 0, 'yes'],
            [0, 0, 0, 0, 'no'],
            [1, 0, 0, 0, 'no'],
            [1, 0, 0, 1, 'no'],
            [1, 1, 1, 1, 'yes'],
            [1, 0, 1, 2, 'yes'],
            [1, 0, 1, 2, 'yes'],
            [2, 0, 1, 2, 'yes'],
            [2, 0, 1, 1, 'yes'],
            [2, 1, 0, 1, 'yes'],
            [2, 1, 0, 2, 'yes'],
            [2, 0, 0, 0, 'no']]
    #分类属性
    labels=['年龄','有工作','有自己的房子','信贷情况']
    #返回数据集和分类属性
    return dataSet,labels


"""
函数说明：计算给定数据集的经验熵（香农熵）
Parameters：
    dataSet：数据集
Returns：
    shannonEnt：经验熵
Modify：
    2018-03-12

"""
def calcShannonEnt(dataSet):
    #返回数据集行数
    numEntries=len(dataSet)
    #保存每个标签（label）出现次数的字典
    labelCounts={}
    #对每组特征向量进行统计
    for featVec in dataSet:
        currentLabel=featVec[-1]                     #提取标签信息
        if currentLabel not in labelCounts.keys():   #如果标签没有放入统计次数的字典，添加进去
            labelCounts[currentLabel]=0
        labelCounts[currentLabel]+=1                 #label计数

    shannonEnt=0.0                                   #经验熵
    #计算经验熵
    for key in labelCounts:
        prob=float(labelCounts[key])/numEntries      #选择该标签的概率
        shannonEnt-=prob*log(prob,2)                 #利用公式计算
    return shannonEnt                                #返回经验熵
```

```
"""
函数说明：计算给定数据集的经验熵（香农熵）
Parameters：
    dataSet：数据集
Returns：
    shannonEnt：信息增益最大特征的索引值
Modify：
    2018-03-12

"""


def chooseBestFeatureToSplit(dataSet):
    #特征数量
    numFeatures = len(dataSet[0]) - 1
    #计数数据集的香农熵
    baseEntropy = calcShannonEnt(dataSet)
    #信息增益
    bestInfoGain = 0.0
    #最优特征的索引值
    bestFeature = -1
    #遍历所有特征
    for i in range(numFeatures):
        # 获取dataSet的第i个所有特征
        featList = [example[i] for example in dataSet]
        #创建set集合{}，元素不可重复
        uniqueVals = set(featList)
        #经验条件熵
        newEntropy = 0.0
        #计算信息增益
        for value in uniqueVals:
            #subDataSet划分后的子集
            subDataSet = splitDataSet(dataSet, i, value)
            #计算子集的概率
            prob = len(subDataSet) / float(len(dataSet))
            #根据公式计算经验条件熵
            newEntropy += prob * calcShannonEnt((subDataSet))
        #信息增益
        infoGain = baseEntropy - newEntropy
        #打印每个特征的信息增益
        print("第%d个特征的增益为%.3f" % (i, infoGain))
        #计算信息增益
        if (infoGain > bestInfoGain):
            #更新信息增益，找到最大的信息增益
            bestInfoGain = infoGain
            #记录信息增益最大的特征的索引值
            bestFeature = i
            #返回信息增益最大特征的索引值
    return bestFeature

"""
函数说明：按照给定特征划分数据集
Parameters：
    dataSet：待划分的数据集
    axis：划分数据集的特征
    value：需要返回的特征的值
Returns：
    shannonEnt：经验熵
Modify：
    2018-03-12

"""
def splitDataSet(dataSet,axis,value):
    retDataSet=[]
    for featVec in dataSet:
        if featVec[axis]==value:
            reducedFeatVec=featVec[:axis]
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
    return retDataSet


#main函数
if __name__=='__main__':
    dataSet,features=creatDataSet()
    # print(dataSet)
    # print(calcShannonEnt(dataSet))
```

```
    print("最优索引值："+str(chooseBestFeatureToSplit(dataSet)))


第0个特征的增益为0.083
第1个特征的增益为0.324
第2个特征的增益为0.420
第3个特征的增益为0.363
最优索引值：2
```

对比我们自己计算的结果，发现结果完全正确！最优特征的索引值为2，也就是特征A3(有自己的房子)。

决策树的生成和修剪

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，包括经验熵的计算和最优特征的选择。

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，数据集被向下传递到树的分支的下一个结点。在这个结点上，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。

构建决策树的算法有很多，比如C4.5、ID3和CART，这些算法在运行时并不总是在每次划分数据分组时都会消耗特征。

由于特征数目并不是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。

决策树生成算法递归地产生决策树，直到不能继续下去未为止。这样产生的树往往对训练数据的分类很准确，但对未知的测试数据的分类却没有那么准确，即出现过拟合现象。

过拟合的原因在于学习时过多地考虑如何提高对训练数据的正确分类，从而构建出过于复杂的决策树。解决这个问题的办法是考虑决策树的复杂度，对已生成的决策树进行简化。

决策树的构建

1. ID3算法
   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：

1）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。

2）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；

3）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

分析数据：

上面已经求得，特征A3（有自己的房子）的信息增益最大，所以选择A3为根节点的特征

它将训练集D划分为两个子集D1（A3取值为“是”）D2（A3取值为“否”）

由于D1只有同一类的样本点，所以它成为一个叶结点，结点的类标记为“是”。

对D2则需要从特征A1(年龄)，A2(有工作)和A4(信贷情况)中选择新的特征，计算各个特征的信息增益：

$$g(D2, A1) = H(D2) - H(D2|A1) = 0.251$$

$$g(D2, A2) = H(D2) - H(D2|A2) = 0.918$$

$$g(D2, A3) = H(D2) - H(D2|A3) = 0.474$$

根据计算，选择信息增益最大的A2作为节点的特征，由于其有两个取值可能，所以引出两个子节点：

①对应“是”（有工作），包含三个样本，属于同一类，所以是一个叶子节点，类标记为“是”

②对应“否”（无工作），包含六个样本，输入同一类，所以是一个叶子节点，类标记为“否”

这样就生成一个决策树，该树只用了两个特征（有两个内部节点），生成的决策树如下图所示：

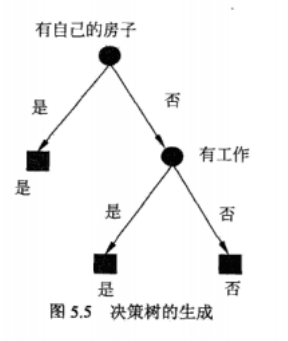

图 5.5　决策树的生成

id3算法代码：

```
from math import log
import operator

"""
函数说明：计算给定数据集的经验熵（香农熵）
Parameters：
    dataSet：数据集
Returns：
    shannonEnt：经验熵
Modify：
    2018-03-12

"""
def calcShannonEnt(dataSet):
    #返回数据集行数
    numEntries=len(dataSet)
    #保存每个标签（label）出现次数的字典
    labelCounts={}
    #对每组特征向量进行统计
    for featVec in dataSet:
        currentLabel=featVec[-1]                     #提取标签信息
        if currentLabel not in labelCounts.keys():   #如果标签没有放入统计次数的字典，添加进去
            labelCounts[currentLabel]=0
        labelCounts[currentLabel]+=1                 #label计数

    shannonEnt=0.0                                   #经验熵
    #计算经验熵
    for key in labelCounts:
        prob=float(labelCounts[key])/numEntries      #选择该标签的概率
        shannonEnt-=prob*log(prob,2)                 #利用公式计算
    return shannonEnt                                #返回经验熵
```

```
"""
函数说明：创建测试数据集
Parameters：无
Returns：
    dataSet：数据集
    labels：分类属性
Modify：
    2018-03-13

"""
def createDataSet():
    # 数据集
    dataSet=[[0, 0, 0, 0, 'no'],
            [0, 0, 0, 1, 'no'],
            [0, 1, 0, 1, 'yes'],
            [0, 1, 1, 0, 'yes'],
            [0, 0, 0, 0, 'no'],
            [1, 0, 0, 0, 'no'],
            [1, 0, 0, 1, 'no'],
            [1, 1, 1, 1, 'yes'],
            [1, 0, 1, 2, 'yes'],
            [1, 0, 1, 2, 'yes'],
            [2, 0, 1, 2, 'yes'],
            [2, 0, 1, 1, 'yes'],
            [2, 1, 0, 1, 'yes'],
            [2, 1, 0, 2, 'yes'],
            [2, 0, 0, 0, 'no']]
    #分类属性
    labels=['年龄','有工作','有自己的房子','信贷情况']
    #返回数据集和分类属性
    return dataSet,labels

"""
函数说明：按照给定特征划分数据集

Parameters：
    dataSet:待划分的数据集
    axis：划分数据集的特征
    value：需要返回的特征值
Returns：
    无
Modify：
    2018-03-13

"""
def splitDataSet(dataSet,axis,value):
    #创建返回的数据集列表
    retDataSet=[]
    #遍历数据集
    for featVec in dataSet:
        if featVec[axis]==value:
            #去掉axis特征
            reduceFeatVec=featVec[:axis]
            #将符合条件的添加到返回的数据集
            reduceFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reduceFeatVec)
    #返回划分后的数据集
    return retDataSet

"""
函数说明：计算给定数据集的经验熵（香农熵）
Parameters：
    dataSet：数据集
Returns：
    shannonEnt：信息增益最大特征的索引值
Modify：
    2018-03-13

"""


def chooseBestFeatureToSplit(dataSet):
    #特征数量
    numFeatures = len(dataSet[0]) - 1
    #计数数据集的香农熵
    baseEntropy = calcShannonEnt(dataSet)
    #信息增益
    bestInfoGain = 0.0
```

```
    #最优特征的索引值
    bestFeature = -1
    #遍历所有特征
    for i in range(numFeatures):
        # 获取dataSet的第i个所有特征
        featList = [example[i] for example in dataSet]
        #创建set集合{}，元素不可重复
        uniqueVals = set(featList)
        #经验条件熵
        newEntropy = 0.0
        #计算信息增益
        for value in uniqueVals:
            #subDataSet划分后的子集
            subDataSet = splitDataSet(dataSet, i, value)
            #计算子集的概率
            prob = len(subDataSet) / float(len(dataSet))
            #根据公式计算经验条件熵
            newEntropy += prob * calcShannonEnt((subDataSet))
        #信息增益
        infoGain = baseEntropy - newEntropy
        #打印每个特征的信息增益
        print("第%d个特征的增益为%.3f" % (i, infoGain))
        #计算信息增益
        if (infoGain > bestInfoGain):
            #更新信息增益，找到最大的信息增益
            bestInfoGain = infoGain
            #记录信息增益最大的特征的索引值
            bestFeature = i
            #返回信息增益最大特征的索引值
    return bestFeature

"""
函数说明：统计classList中出现次数最多的元素（类标签）
Parameters：
    classList：类标签列表
Returns：
    sortedClassCount[0][0]：出现次数最多的元素（类标签）
Modify：
    2018-03-13

"""
def majorityCnt(classList):
    classCount={}
    #统计classList中每个元素出现的次数
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote]=0
            classCount[vote]+=1
        #根据字典的值降序排列
        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：训练数据集
    labels：分类属性标签
    featLabels：存储选择的最优特征标签
Returns：
    myTree：决策树
Modify：
    2018-03-13

"""
def createTree(dataSet,labels,featLabels):
    #取分类标签（是否放贷：yes or no）
    classList=[example[-1] for example in dataSet]
    #如果类别完全相同，则停止继续划分
    if classList.count(classList[0])==len(classList):
        return classList[0]
    #遍历完所有特征时返回出现次数最多的类标签
    if len(dataSet[0])==1:
        return majorityCnt(classList)
    #选择最优特征
    bestFeat=chooseBestFeatureToSplit(dataSet)
    #最优特征的标签
    bestFeatLabel=labels[bestFeat]
```

```
    featLabels.append(bestFeatLabel)
    #根据最优特征的标签生成树
    myTree={bestFeatLabel:{}}
    #删除已经使用的特征标签
    del(labels[bestFeat])
    #得到训练集中所有最优特征的属性值
    featValues=[example[bestFeat] for example in dataSet]
    #去掉重复的属性值
    uniqueVls=set(featValues)
    #遍历特征，创建决策树
    for value in uniqueVls:
        myTree[bestFeatLabel][value]=createTree(splitDataSet(dataSet,bestFeat,value),
                                                labels,featLabels)
    return myTree

if __name__=='__main__':
    dataSet,labels=createDataSet()
    featLabels=[]
    myTree=createTree(dataSet,labels,featLabels)
    print(myTree)

结果
第0个特征的增益为0.083
第1个特征的增益为0.324
第2个特征的增益为0.420
第3个特征的增益为0.363
第0个特征的增益为0.252
第1个特征的增益为0.918
第2个特征的增益为0.474
{'有自己的房子': {0: {'有工作': {0: 'no', 1: 'yes'}}, 1: 'yes'}}
```

决策树的剪枝

决策树生成算法递归的产生决策树，直到不能继续下去为止，这样产生的树往往对训练数据的分类很准确，但对未知测试数据的分类缺没有那么精确，即会出现过拟合现象。

过拟合产生的原因在于在学习时过多的考虑如何提高对训练数据的正确分类，从而构建出过于复杂的决策树，解决方法是考虑决策树的复杂度，对已经生成的树进行简化。

剪枝（pruning）：从已经生成的树上裁掉一些子树或叶节点，并将其根节点或父节点作为新的叶子节点，从而简化分类树模型。

实现方式：极小化决策树整体的损失函数或代价函数来实现

决策树学习的损失函数定义为：

$$C_\alpha(T)=\sum_{t=1}^{|T|}N_tH_t(T)+\alpha|T|$$

其中：

参数	意义
T	表示这棵子树的叶子节点，
$H_t(T)$	表示第t个叶子的熵，[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传(img-QHgonDQQ-1663157039225)(//img-blog.csdn.net/20180314091812955?watermark/2/text/Ly9ibG9nLmNzZG4ubmV0L2ppYW95YW5nd20=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70)]
N_t	表示该叶子所含的训练样例的个数，
α	惩罚系数，
$\|T\|$	表示子树的叶子节点的个数。

因为：

其中经验熵为

$$H_t(T)=-\sum_k \frac{N_{tk}}{N_t}\log\frac{N_{tk}}{N_t} \tag{5.12}$$

在损失函数中，将式 (5.11) 右端的第 1 项记作

$$C(T)=\sum_{t=1}^{|T|} N_t H_t(T)=-\sum_{t=1}^{|T|}\sum_{k=1}^{K} N_{tk}\log\frac{N_{tk}}{N_t} \tag{5.13}$$

所以有：$C_\alpha(T)=C(T)+\alpha|T|$

其中：

参数	意义
$C(T)$	表示模型对训练数据的预测误差，即模型与训练数据的拟合程度；
$\|T\|$	表示模型复杂度
α	参数$\alpha >= 0$控制两者之间的影响，较大的α促使选择较简单的模型（树），较小的α促使选择较复杂的模型（树），$\alpha = 0$意味着只考虑模型与训练数据的拟合程度，不考虑模型的复杂度。

剪枝就是当α确定时，选择损失函数最小的模型，即损失函数最小的子树。

剪枝就是当α 确定时，选择损失函数最小的模型，即损失函数最小的子树。

当α值确定时，子树越大，往往与训练数据的拟合越好，但是模型的复杂度越高；
子树越小，模型的复杂度就越低，但是往往与训练数据的拟合不好
损失函数正好表示了对两者的平衡。

损失函数认为对于每个分类终点（叶子节点）的不确定性程度就是分类的损失因子，而叶子节点的个数是模型的复杂程度，作为惩罚项，损失函数的第一项是样本的训练误差，第二项是模型的复杂度。

如果一棵子树的损失函数值越大，说明这棵子树越差，因此我们希望让每一棵子树的损失函数值尽可能得小，损失函数最小化就是用正则化的极大似然估计进行模型选择的过程。

决策树算法很容易过拟合（overfitting），剪枝算法就是用来防止决策树过拟合，提高泛华性能的方法。

剪枝分为预剪枝与后剪枝。

预剪枝是指在决策树的生成过程中，对每个节点在划分前先进行评估，若当前的划分不能带来泛化性能的提升，则停止划分，并将当前节点标记为叶节点。

后剪枝是指先从训练集生成一颗完整的决策树，然后自底向上对非叶节点进行考察，若将该节点对应的子树替换为叶节点，能带来泛化性能的提升，则将该子树替换为叶节点。

那么怎么来判断是否带来泛化性能的提升那？最简单的就是留出法，即预留一部分数据作为验证集来进行性能评估。

决策树可视化

```
from math import log
import operator
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt
"""
函数说明：计算给定数据集的经验熵（香农熵）
Parameters：
    dataSet：数据集
Returns：
    shannonEnt：经验熵
Modify：
    2018-03-12

"""
def calcShannonEnt(dataSet):
    #返回数据集行数
```

```
    numEntries=len(dataSet)
    #保存每个标签（label）出现次数的字典
    labelCounts={}
    #对每组特征向量进行统计
    for featVec in dataSet:
        currentLabel=featVec[-1]                        #提取标签信息
        if currentLabel not in labelCounts.keys():   #如果标签没有放入统计次数的字典，添加进去
            labelCounts[currentLabel]=0
        labelCounts[currentLabel]+=1                    #label计数

    shannonEnt=0.0                                      #经验熵
    #计算经验熵
    for key in labelCounts:
        prob=float(labelCounts[key])/numEntries      #选择该标签的概率
        shannonEnt-=prob*log(prob,2)                    #利用公式计算
    return shannonEnt                                   #返回经验熵

"""
函数说明：创建测试数据集
Parameters：无
Returns：
    dataSet：数据集
    labels：分类属性
Modify：
    2018-03-13

"""
def createDataSet():
    # 数据集
    dataSet=[[0, 0, 0, 0, 'no'],
            [0, 0, 0, 1, 'no'],
            [0, 1, 0, 1, 'yes'],
            [0, 1, 1, 0, 'yes'],
            [0, 0, 0, 0, 'no'],
            [1, 0, 0, 0, 'no'],
            [1, 0, 0, 1, 'no'],
            [1, 1, 1, 1, 'yes'],
            [1, 0, 1, 2, 'yes'],
            [1, 0, 1, 2, 'yes'],
            [2, 0, 1, 2, 'yes'],
            [2, 0, 1, 1, 'yes'],
            [2, 1, 0, 1, 'yes'],
            [2, 1, 0, 2, 'yes'],
            [2, 0, 0, 0, 'no']]
    #分类属性
    labels=['年龄','有工作','有自己的房子','信贷情况']
    #返回数据集和分类属性
    return dataSet,labels

"""
函数说明：按照给定特征划分数据集

Parameters：
    dataSet:待划分的数据集
    axis：划分数据集的特征
    value：需要返回的特征值
Returns：
    无
Modify：
    2018-03-13

"""
def splitDataSet(dataSet,axis,value):
    #创建返回的数据集列表
    retDataSet=[]
    #遍历数据集
    for featVec in dataSet:
        if featVec[axis]==value:
            #去掉axis特征
            reduceFeatVec=featVec[:axis]
            #将符合条件的添加到返回的数据集
            reduceFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reduceFeatVec)
    #返回划分后的数据集
    return retDataSet

"""
函数说明：计算给定数据集的经验熵（香农熵）
```

```
Parameters：
    dataSet：数据集
Returns：
    shannonEnt：信息增益最大特征的索引值
Modify：
    2018-03-13

"""


def chooseBestFeatureToSplit(dataSet):
    #特征数量
    numFeatures = len(dataSet[0]) - 1
    #计数数据集的香农熵
    baseEntropy = calcShannonEnt(dataSet)
    #信息增益
    bestInfoGain = 0.0
    #最优特征的索引值
    bestFeature = -1
    #遍历所有特征
    for i in range(numFeatures):
        # 获取dataSet的第i个所有特征
        featList = [example[i] for example in dataSet]
        #创建set集合{}，元素不可重复
        uniqueVals = set(featList)
        #经验条件熵
        newEntropy = 0.0
        #计算信息增益
        for value in uniqueVals:
            #subDataSet划分后的子集
            subDataSet = splitDataSet(dataSet, i, value)
            #计算子集的概率
            prob = len(subDataSet) / float(len(dataSet))
            #根据公式计算经验条件熵
            newEntropy += prob * calcShannonEnt((subDataSet))
        #信息增益
        infoGain = baseEntropy - newEntropy
        #打印每个特征的信息增益
        print("第%d个特征的增益为%.3f" % (i, infoGain))
        #计算信息增益
        if (infoGain > bestInfoGain):
            #更新信息增益，找到最大的信息增益
            bestInfoGain = infoGain
            #记录信息增益最大的特征的索引值
            bestFeature = i
            #返回信息增益最大特征的索引值
    return bestFeature

"""
函数说明：统计classList中出现次数最多的元素（类标签）
Parameters：
    classList：类标签列表
Returns：
    sortedClassCount[0][0]：出现次数最多的元素（类标签）
Modify：
    2018-03-13

"""
def majorityCnt(classList):
    classCount={}
    #统计classList中每个元素出现的次数
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote]=0
            classCount[vote]+=1
        #根据字典的值降序排列
        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：训练数据集
    labels：分类属性标签
    featLabels：存储选择的最优特征标签
Returns：
    myTree：决策树
```

```
Modify：
    2018-03-13

"""
def createTree(dataSet,labels,featLabels):
    #取分类标签（是否放贷：yes or no）
    classList=[example[-1] for example in dataSet]
    #如果类别完全相同，则停止继续划分
    if classList.count(classList[0])==len(classList):
        return classList[0]
    #遍历完所有特征时返回出现次数最多的类标签
    if len(dataSet[0])==1:
        return majorityCnt(classList)
    #选择最优特征
    bestFeat=chooseBestFeatureToSplit(dataSet)
    #最优特征的标签
    bestFeatLabel=labels[bestFeat]
    featLabels.append(bestFeatLabel)
    #根据最优特征的标签生成树
    myTree={bestFeatLabel:{}}
    #删除已经使用的特征标签
    del(labels[bestFeat])
    #得到训练集中所有最优特征的属性值
    featValues=[example[bestFeat] for example in dataSet]
    #去掉重复的属性值
    uniqueVls=set(featValues)
    #遍历特征，创建决策树
    for value in uniqueVls:
        myTree[bestFeatLabel][value]=createTree(splitDataSet(dataSet,bestFeat,value),
                                                labels,featLabels)
    return myTree

"""
函数说明：获取决策树叶子节点的数目

Parameters：
    myTree：决策树
Returns：
    numLeafs：决策树的叶子节点的数目
Modify：
    2018-03-13

"""

def getNumLeafs(myTree):
    numLeafs=0
    firstStr=next(iter(myTree))
    secondDict=myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
            numLeafs+=getNumLeafs(secondDict[key])
        else: numLeafs+=1
    return numLeafs

"""
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

Modify:
    2018-03-13
"""
def getTreeDepth(myTree):
    maxDepth = 0                                                #初始化决策树深度
    firstStr = next(iter(myTree))                               #python3中myTree.keys()返回的是dict_keys,不在是list,所以不能使用myTree.keys()[0
    secondDict = myTree[firstStr]                               #获取下一个字典
    for key in secondDict.keys():
        if type(secondDict[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 = 1 + getTreeDepth(secondDict[key])
        else:   thisDepth = 1
        if thisDepth > maxDepth: maxDepth = thisDepth           #更新层数
    return maxDepth

"""
函数说明:绘制结点
```

```
Parameters:
    nodeTxt - 结点名
    centerPt - 文本位置
    parentPt - 标注的箭头位置
    nodeType - 结点格式
Returns:
    无
Modify:
    2018-03-13
"""
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    arrow_args = dict(arrowstyle="<-")                                          #定义箭头格式
    font = FontProperties(fname=r"c:\windows\fonts\simsun.ttc", size=14)        #设置中文字体
    createPlot.ax1.annotate(nodeTxt, xy=parentPt,  xycoords='axes fraction',    #绘制结点
        xytext=centerPt, textcoords='axes fraction',
        va="center", ha="center", bbox=nodeType, arrowprops=arrow_args, FontProperties=font)

"""
函数说明:标注有向边属性值

Parameters:
    cntrPt、parentPt - 用于计算标注位置
    txtString - 标注的内容
Returns:
    无
Modify:
    2018-03-13
"""
def plotMidText(cntrPt, parentPt, txtString):
    xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]                                  #计算标注位置
    yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]
    createPlot.ax1.text(xMid, yMid, txtString, va="center", ha="center", rotation=30)

"""
函数说明:绘制决策树

Parameters:
    myTree - 决策树(字典)
    parentPt - 标注的内容
    nodeTxt - 结点名
Returns:
    无
Modify:
    2018-03-13
"""
def plotTree(myTree, parentPt, nodeTxt):
    decisionNode = dict(boxstyle="sawtooth", fc="0.8")                                #设置结点格式
    leafNode = dict(boxstyle="round4", fc="0.8")                                     #设置叶结点格式
    numLeafs = getNumLeafs(myTree)                                                   #获取决策树叶结点数目，决定了树的宽度
    depth = getTreeDepth(myTree)                                                     #获取决策树层数
    firstStr = next(iter(myTree))                                                     #下个字典
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)    #中心位置
    plotMidText(cntrPt, parentPt, nodeTxt)                                            #标注有向边属性值
    plotNode(firstStr, cntrPt, parentPt, decisionNode)                                #绘制结点
    secondDict = myTree[firstStr]                                                    #下一个字典，也就是继续绘制子结点
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD                                 #y偏移
    for key in secondDict.keys():
        if type(secondDict[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
            plotTree(secondDict[key],cntrPt,str(key))                                #不是叶结点，递归调用继续绘制
        else:                                                                        #如果是叶结点，绘制叶结点，并标注有向边属性值
            plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW
            plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff), cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD

"""
函数说明:创建绘制面板

Parameters:
    inTree - 决策树(字典)
Returns:
    无
Modify:
    2018-03-13
"""
def createPlot(inTree):
    fig = plt.figure(1, facecolor='white')#创建fig
```

```
    fig.clf()#清空fig
    axprops = dict(xticks=[], yticks=[])
    createPlot.ax1 = plt.subplot(111, frameon=False, **axprops)#去掉x、y轴
    plotTree.totalW = float(getNumLeafs(inTree))#获取决策树叶结点数目
    plotTree.totalD = float(getTreeDepth(inTree))#获取决策树层数
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0#x偏移
    plotTree(inTree, (0.5,1.0), '')#绘制决策树
    plt.show()#显示绘制结果

if __name__ == '__main__':
    dataSet, labels = createDataSet()
    featLabels = []
    myTree = createTree(dataSet, labels, featLabels)
    print(myTree)
    createPlot(myTree)

if __name__=='__main__':
    dataSet,labels=createDataSet()
    featLabels=[]
    myTree=createTree(dataSet,labels,featLabels)
    print(myTree)
```

ID3、C4.5、CART的区别
这三个是非常著名的决策树算法。简单粗暴来说，ID3 使用信息增益作为选择特征的准则；C4.5 使用信息增益比作为选择特征的准则；CART 使用 Gini 指数作为选择特征的准则。

一、ID3
熵表示的是数据中包含的信息量大小。熵越小，数据的纯度越高，也就是说数据越趋于一致，这是我们希望的划分之后每个子节点的样子。

信息增益 = 划分前熵 - 划分后熵。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 。也就是说，用属性 a 来划分训练集，得到的结果中纯度比较高。

ID3 仅仅适用于二分类问题。ID3 仅仅能够处理离散属性。

二、C4.5

C4.5 克服了 ID3 仅仅能够处理离散属性的问题，以及信息增益偏向选择取值较多特征的问题，使用信息增益比来选择特征。信息增益比 = 信息增益 / 划分前熵 选择信息增益比最大的作为最优特征。

C4.5 处理连续特征是先将特征取值排序，以连续两个值中间值作为划分标准。尝试每一种划分，并计算修正后的信息增益，选择信息增益最大的分裂点作为该属性的分裂点。

三、CART

CART 与 ID3，C4.5 不同之处在于 CART 生成的树必须是二叉树。也就是说，无论是回归还是分类问题，无论特征是离散的还是连续的，无论属性取值有多个还是两个，内部节点只能根据属性值进行二分。

CART 的全称是分类与回归树。从这个名字中就应该知道，CART 既可以用于分类问题，也可以用于回归问题。

回归树中，使用平方误差最小化准则来选择特征并进行划分。每一个叶子节点给出的预测值，是划分到该叶子节点的所有样本目标值的均值，这样只是在给定划分的情况下最小化了平方误差。

要确定最优化分，还需要遍历所有属性，以及其所有的取值来分别尝试划分并计算在此种划分情况下的最小平方误差，选取最小的作为此次划分的依据。由于回归树生成使用平方误差最小化准则，所以又叫做最小二乘回归树。

分类树种，使用 Gini 指数最小化准则来选择特征并进行划分；

Gini 指数表示集合的不确定性，或者是不纯度。基尼指数越大，集合不确定性越高，不纯度也越大。这一点和熵类似。另一种理解基尼指数的思路是，基尼指数是为了最小化误分类的概率。

信息增益 vs 信息增益比

之所以引入了信息增益比，是由于信息增益的一个缺点。那就是：信息增益总是偏向于选择取值较多的属性。信息增益比在此基础上增加了一个罚项，解决了这个问题。

Gini 指数 vs 熵

既然这两个都可以表示数据的不确定性，不纯度。那么这两个有什么区别那？

Gini 指数的计算不需要对数运算，更加高效；
Gini 指数更偏向于连续属性，熵更偏向于离散属性。

优点：

1.易于理解和解释，决策树可以可视化。
2.几乎不需要数据预处理。其他方法经常需要数据标准化，创建虚拟变量和删除缺失值。决策树还不支持缺失值。
3.使用树的花费（例如预测数据）是训练数据点(data points)数量的对数。
4.可以同时处理数值变量和分类变量。其他方法大都适用于分析一种变量的集合。
5.可以处理多值输出变量问题。

缺点：

1.决策树学习可能创建一个过于复杂的树，并不能很好的预测数据。也就是过拟合。修剪机制（现在不支持），设置一个叶子节点需要的最小样本数量，或者数的最大深度，可以避免过拟合。
2.决策树可能是不稳定的，因为即使非常小的变异，可能会产生一颗完全不同的树。这个问题通过decision trees with an ensemble来缓解。