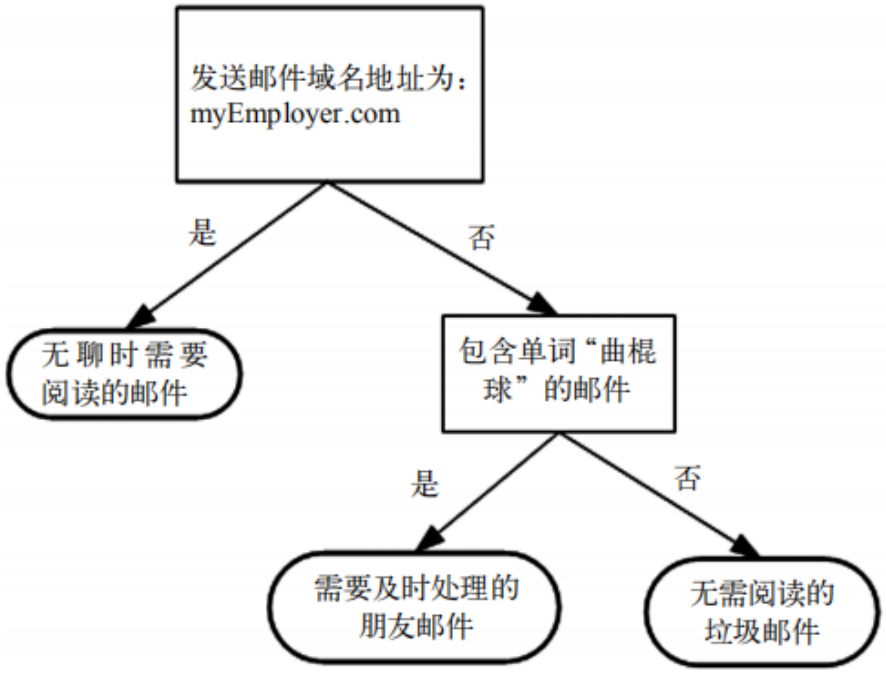
什么是决策树？

- 示例：



流程图就是一个决策树，长方形代表判断模块 （decision block），椭圆形代表终止模块（terminating block），表示已经得出结论，可以终止运行。 从判断模块引出的左右箭头称作分支（branch），它可以到达另一个判断模块或者终止模块。图中构造了一个假想的邮件分类系统，它首先检测发送邮件域名地址。如果地址为 myEmployer.com，则将其放在分类“无聊时需要阅读的邮件”中。如果邮件不是来自这个域名， 则检查邮件内容里是否包含单词曲棍球，如果包含则将邮件归类到“需要及时处理的朋友邮件”， 如果不包含则将邮件归类到“无需阅读的垃圾邮件”

- 理解的决策树：简单理解就是if elif else 的语句，判断判断再判断，直到能得到一个比较满意的label

构建决策树

- 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问题

  适用数据类型：数值型和标称型

- 创建分支的伪代码

```
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createBranch并增加返回结果到分支节点中
return 分支节点
```

- 流程:

```
收集数据：可以使用任何方法。
准备数据：树构造算法 (这里使用的是ID3算法，只适用于标称型数据，这就是为什么数值型数据必须离散化。 还有其他的树构造算法，比如CART)
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。
```

- 举例模板:

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

熵及其有关概念

- 熵(entropy):指的是群体的混乱程度，我们的决策树的要求是在特征下将条件熵降到最低

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

  其中p(xi)是选择该分类的概率

- 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

- 信息增益：在划分数据集之前之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

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

- 条件熵:这里我只给一个公式,主要是一个大哥讲的更好:网址

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

- 香农熵(信息熵的计算)：

```
from math import log
def calcShannonEnt(dataSet):
    # 求list的长度，表示计算参与训练的数据量
    numEntries = len(dataSet)
    # 计算分类标签label出现的次数
    labelCounts = {}
    # the the number of unique elements and their
occurrence
    for featVec in dataSet:
        # 将当前实例的标签存储，即每一行数据的最后一个数据代表的是标
签
        currentLabel = featVec[-1]
        # 为所有可能的分类创建字典，如果当前的键值不存在，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。
        if currentLabel not in labelCounts.keys():
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1

    # 对于 label 标签的占比，求出 label 标签的香农熵
    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
            # 计算香农熵，以 2 为底求对数
            shannonEnt -= prob * log(prob, 2)
        return shannonEnt
```

- 基尼不纯度：

$$I_G(f) = \sum_{i=1}^{m} f_i(1-f_i) = \sum_{i=1}^{m}(f_i - f_i^2) = \sum_{i=1}^{m} f_i - \sum_{i=1}^{m} f_i^2 = 1 - \sum_{i=1}^{m} f_i^2$$

讲解案例：

一个随机事件X，P(X=0)= 0.5 ,P(X=1)=0.5

```
那么基尼不纯度就为   P(X=0)*(1 - P(X=0)) +  P(X=1)*(1 - P(X=1))  = 0.5
```

```
一个随机事件Y ，P(Y=0)= 0.1 ,P(Y=1)=0.9

那么基尼不纯度就为P(Y=0)*(1 - P(Y=0)) +   P(Y=1)*(1 -P(Y=1))  = 0.18

很明显 X比Y更混乱，因为两个都为0.5 很难判断哪个发生。而Y就确定得多，Y=1发生的概率很大。而基尼不纯度也就越小。
```

结论：

（1）基尼不纯度可以作为 衡量系统混乱程度的 标准；

（2）基尼不纯度越小，纯度越高，集合的有序程度越高，分类的效果越好；

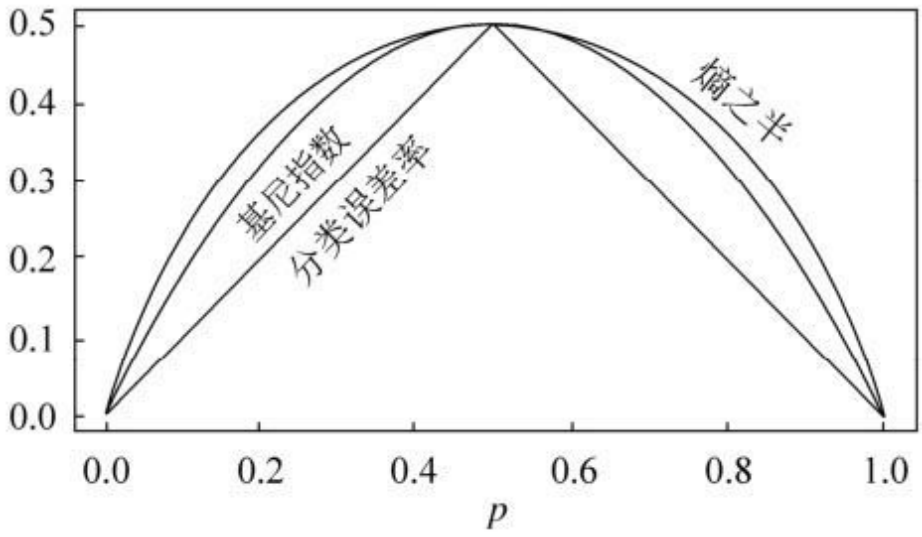
（3）基尼不纯度为 0 时，表示集合类别一致；

（4）在决策树中，比较基尼不纯度的大小可以选择更好的决策条件（子节点）

```
# 示例代码：
my_data = [['fan', 'C', 'yes', 32, 'None'],
           ['fang', 'U', 'yes', 23, 'Premium'],
           ['ming', 'F', 'no', 28, 'Basic']]


# 计算每一行数据的可能数量
def uniqueCounts(rows):
```

```
    results = {}
    for row in rows:
        # 对最后一列的值计算
        # r = row[len(row) - 1]
        # 对倒数第三的值计算，也就是yes 和no 的一列
        r = row[len(row) - 3]
        if r not in results: results[r] = 0
        results[r] += 1
    return results


# 基尼不纯度样例
def giniImpurityExample(rows):
    total = len(rows)
    print(total)
    counts = uniqueCounts(rows)
    print(counts)
    imp = 0
    for k1 in counts:
        p1 = float(counts[k1]) / total
        print(counts[k1])
        for k2 in counts:
            if k1 == k2: continue
            p2 = float(counts[k2]) / total
            imp += p1 * p2
    return imp


gini = giniImpurityExample(my_data)
print('gini Impurity is %s' % gini)
```

- 总结：



构建决策树

- 数据集：

```
def createDataSet():
    dataSet = [[1, 1, 'yes'],
               [1, 1, 'yes'],
               [1, 0, 'no'],
               [0, 1, 'no'],
               [0, 1, 'no']]
    labels = ['no surfacing', 'flippers']
    return dataSet, labels
myDat,labels = createDataSet()
```

这里我们使用特征数值化，label标称化，true是1，false是0，label是yes 或者no

结果：

```
In  [4]: myDat
Out[4]: [[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]

In  [5]: labels
Out[5]: ['no surfacing', 'flippers']
```

运行熵计算函数

```
In  [6]: print(calcShannonEnt(myDat))
         0.9709505944546686

In  [7]: myDat[0][-1]='maybe'

In  [8]: calcShannonEnt(myDat)
Out[8]: 0.9709505944546686
```

按照给定特征划分数据集

- 代码：

```
def splitDataSet(dataSet, index, value):
    """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
    Args:
        dataSet 数据集                 待划分的数据集
        index 表示每一行的index列      划分数据集的特征
        value 表示index列对应的value值  需要返回的特征的值。
    Returns:
        index列为value的数据集【该数据集需要排除index列】
    """
    retDataSet = []
    for featVec in dataSet:
        # index列为value的数据集【该数据集需要排除index列】
        # 判断index列的值是否为value
        if featVec[index] == value:
            # chop out index used for splitting
```

```
            # [:index]表示前index行，即若 index 为2，就是取
featVec 的前 index 行
            reducedFeatVec = featVec[:index]
            '''
            请百度查询一下： extend和append的区别
            music_media.append(object) 向列表中添加一个对象
object
            music_media.extend(sequence) 把一个序列seq的内容
添加到列表中（跟 += 在list运用类似， music_media += sequence)
            1、使用append的时候，是将object看作一个对象，整体打包
添加到music_media对象中。
            2、使用extend的时候，是将sequence看作一个序列，将这个
序列和music_media序列合并，并放在其后面。
            music_media = []
            music_media.extend([1,2,3])
            print music_media
            #结果:
            #[1, 2, 3]

            music_media.append([4,5,6])
            print music_media
            #结果:
            #[1, 2, 3, [4, 5, 6]]

            music_media.extend([7,8,9])
            print music_media
            #结果:
            #[1, 2, 3, [4, 5, 6], 7, 8, 9]
            '''
            reducedFeatVec.extend(featVec[index+1:])
            # [index+1:]表示从跳过 index 的 index+1行，取接下
来的数据
            # 收集结果值 index列为value的行【该行需要排除index
列】
            retDataSet.append(reducedFeatVec)
    return retDataSet
```

- 运行示例：

```
In [10]: # 看例子说话
         myDat,labels = createDataSet()

In [11]: myDat
Out[11]: [[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]

In [12]: splitDataSet(myDat,0,1)#这里就是查找第0列等于1的除去第0列的数据
Out[12]: [[1, 'yes'], [1, 'yes'], [0, 'no']]

In [13]: splitDataSet(myDat,0,0)
Out[13]: [[1, 'no'], [1, 'no']]
```

其实上面划分数据集的，就是将数据中指定的索引等于指定值的数据提取出来

选择最好的数据集划分方式

- 划分代码:

```
#选择最好的数据集划分方式
def chooseBestFeatureToSplit(dataSet):
    """chooseBestFeatureToSplit(选择最好的特征)

    Args:
        dataSet 数据集
    Returns:
        bestFeature 最优的特征列
    """
    # 求第一行有多少列的 Feature, 最后一列是label列嘛
    numFeatures = len(dataSet[0]) - 1
    # 数据集的原始信息熵
    baseEntropy = calcShannonEnt(dataSet)
    # 最优的信息增益值, 和最优的Featurn编号
    bestInfoGain, bestFeature = 0.0, -1
    # iterate over all the features
    for i in range(numFeatures):
        # create a list of all the examples of this
feature
        # 获取对应的feature下的所有数据
        featList = [example[i] for example in dataSet]
        # get a set of unique values
        # 获取剔重后的集合，使用set对list数据进行去重
        uniqueVals = set(featList)
        # 创建一个临时的信息熵
        newEntropy = 0.0
        # 遍历某一列的value集合，计算该列的信息熵
        # 遍历当前特征中的所有唯一属性值，对每个唯一属性值划分一次数
据集，计算数据集的新熵值，并对所有唯一特征值得到的熵求和。
        for value in uniqueVals:
            subDataSet = splitDataSet(dataSet, i, value)
            # 计算概率
            prob = len(subDataSet)/float(len(dataSet))
            # 计算条件熵
            newEntropy += prob *
calcShannonEnt(subDataSet)
        # gain[信息增益]: 划分数据集前后的信息变化， 获取信息熵最大
的值
        # 信息增益是熵的减少或者是数据无序度的减少。最后，比较所有特
征中的信息增益，返回最好特征划分的索引值。
        infoGain = baseEntropy - newEntropy
        print('infoGain=', infoGain, 'bestFeature=', i,
baseEntropy, newEntropy)
        if (infoGain > bestInfoGain):
            bestInfoGain = infoGain
            bestFeature = i
    return bestFeature
```

这个代码中的条件熵的计算我自己认为是一个期望熵，就是按照这个特征分下去，总体的期望

- 运行示例：

```
In [10]: # 看例子说话
         myDat,labels = createDataSet()

In [11]: myDat
Out[11]: [[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]

In [12]: splitDataSet(myDat,0,1)#这里就是查找第0列等于1的除去第0列的数据
Out[12]: [[1, 'yes'], [1, 'yes'], [0, 'no']]

In [13]: splitDataSet(myDat,0,0)
Out[13]: [[1, 'no'], [1, 'no']]
```

解释：其实就是按照第0个特征来分，总体的信息增益最大，也就是群体有序程度更高

递归创建决策树

- 投票机制：vote，这是没有剪枝情况下，最坏情况下使用的投票机制，也就是没有找到完全划分的方式

```
def majorityCnt(classList):
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
        else:
            classCount[vote] += 1
        sortedClassCount =
sorted(classCount.item(),key=lambda x:x[1],reverse = True)
        return sortedClassCount[0][0]
```

- 创建树的代码

```
def createTree(dataSet, labels):
    classList = [example[-1] for example in dataSet]
    # 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。
    # count() 函数是统计括号中的值在list中出现的次数
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    # 如果数据集只有1列，那么最初出现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)

    # 选择最优的列，得到最优列对应的label含义
    bestFeat = chooseBestFeatureToSplit(dataSet)
```

```
    # 获取label的名称
    bestFeatLabel = labels[bestFeat]
    # 初始化myTree
    myTree = {bestFeatLabel: {}}
    # 注：labels列表是可变对象，在PYTHON函数中作为参数时传址引用，能够被全局修改
    # 所以这行代码导致函数外的同名变量被删除了元素，造成例句无法执行，提示'no surfacing' is not in list
    del(labels[bestFeat])
    # 取出最优列，然后它的branch做分类
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        # 求出剩余的标签label
        subLabels = labels[:]
        # 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)
        # print 'myTree', value, myTree
    return myTree
```

这里比较难理解，自己debug一下

- 运行示例：

```
In [20]: labels_copy = labels[:]
         labels_copy2 = labels[:]
         mytree = createTree(myDat,labels_copy)
         tree = createTree(myDat,labels_copy2)

         infoGain= 0.4199730940219749 bestFeature= 0 0.9709505944546686 0.5509775004326937
         infoGain= 0.17095059445466854 bestFeature= 1 0.9709505944546686 0.8
         infoGain= 0.9182958340544896 bestFeature= 0 0.9182958340544896 0.0
         infoGain= 0.4199730940219749 bestFeature= 0 0.9709505944546686 0.5509775004326937
         infoGain= 0.17095059445466854 bestFeature= 1 0.9709505944546686 0.8
         infoGain= 0.9182958340544896 bestFeature= 0 0.9182958340544896 0.0

In [20]: mytree

Out[20]: {'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}

In [21]: print(labels)

         ['no surfacing', 'flippers']
```

- 决策树测试代码：

```
def classify(inputTree, featLabels, testVec):
    '''
    函数功能：
            对测试实例进行分类
    参数说明：
            inputTree__已经训练好的决策树
            featLabels__特征标签类别
            testVec__测试示例
    函数返回：
            分类结果
    '''
```

```
    # python3.x中input.key()[0]返回的是dict_keys，不是list，这里注意区别（书上的代码是python2.x）
    firstStr = list(inputTree.keys())[0]  # 获得决策树第一个节点
    #print(featLabels)
    secondDict = inputTree[firstStr]      # 获取下一个字典
    print(secondDict)
    print(firstStr)
    featIndex = featLabels.index(firstStr)    # 将标签字符串转换为索引(第一个节点所在列的索引)
    for key in secondDict.keys():
        #print(testVec[featIndex])
        if testVec[featIndex] == key:
            if type(secondDict[key]).__name__ == 'dict':
                classLabel = classify(secondDict[key], featLabels, testVec)
            else:
                classLabel = secondDict[key]
    return classLabel
```

- 运行示例

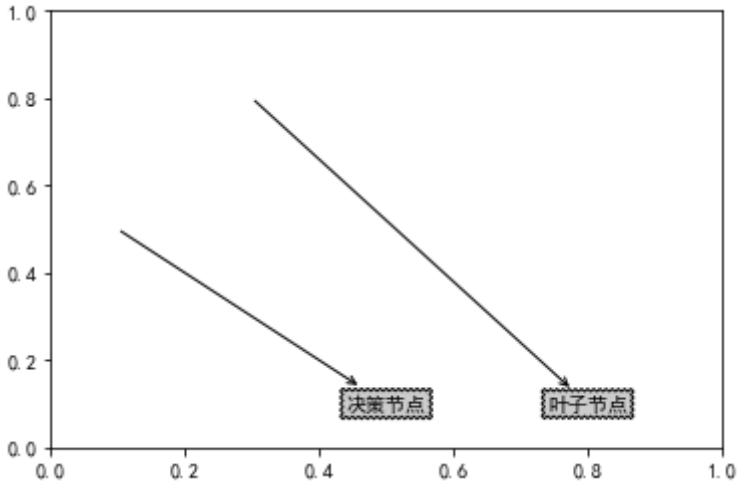
```
In [23]: classify(mytree,labels,[1,0])

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

Out[23]: 'no'
```

matplotlib注解绘制树形图

- matplotlib绘制图像示例

```
#使用matploblib注解绘制树形图
#使用matploblib注解绘制树形图
import matplotlib.pyplot as plt

from pylab import *
mpl.rcParams['font.sans-serif'] = ['SimHei']#中文注释
decisionNode = dict(boxstyle='sawtooth',fc="0.8")
leafNode = dict(boxstyle="round4",fc="0.8")
arrow_args = dict(arrowstyle="<-")
def plotNode(nodeTxt,centerPt,parentPt,nodeType):

 createPlot.axl.annotate(nodeTxt,xy=parentPt,xycoords="axes fraction",xytext=centerPt,textcoords="axes fraction"

,va="center",ha="center",bbox=nodeType,arrowprops=arrow_args)#
https://blog.csdn.net/leaf_zizi/article/details/82886755
def createPlot():
```

```
    fig = plt.figure(1,facecolor="white")
    fig.clf
    createPlot.axl = plt.subplot(111,facecolor="white")
    plotNode(U"决策节点",(0.5,0.1),(0.1,0.5),decisionNode)
    plotNode(U"叶子节点",(0.8,0.1),(0.3,0.8),decisionNode)
    plt.show()
#https://blog.csdn.net/u013038499/article/details/52449768
```



决策树属性的描述

- 叶子节点数目

```
#获得叶子节点数目
def getNumLeafs(myTree):
    '''
    函数功能：
            递归计算叶子节点数目
    函数参数：
            字典形式的决策树
    函数返回：
            叶子节点数目
    '''
    numLeafs = 0
# 初始化叶子节点的数目
    print(list(myTree.keys()))
    firstStr = list(myTree.keys())[0]
 # 获取决策树的第一个节点
    secondDict = myTree[firstStr]
 # 获取决策树的第二个节点
    for key in secondDict.keys():

        if type(secondDict[key]).__name__ == 'dict':
# 若该节点为字典形式
            numLeafs += getNumLeafs(secondDict[key])
# 若为字典，则递归计算新分支叶节点数
        else:
```

```
            numLeafs += 1
 # 若不是字典，则此节点为叶子节点
    return numLeafs
 # 返回叶子节点数目

# 函数测试
# labels = ['no surfacing', 'flippers', 'labels']
# labels_copy2 = labels[:]
# print(myDat)
# print(labels)
# tree = createTree(myDat,labels_copy2)
```

- 树深度

```
#得到树的深度
def getTreeDepth(myTree):
    '''
    函数功能：
            递归计算决策树的深度
    函数参数：
            myTree__字典形式的决策树
    函数返回：
            决策树的最大深度
    '''
    maxDepth = 0

    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:
            thisDepth = 1
        if thisDepth > maxDepth:
            maxDepth = thisDepth
    return maxDepth
#getTreeDepth(mytree)
```

树的标注

- 标注使用详细
- 文本标注

```
#使用文本标注
decisionNode = dict(boxstyle = 'sawtooth', fc = '0.8')
 # 设置中间节点的格式
leafNode = dict(boxstyle = 'round4', fc = '0.8')
 # 设置叶子节点的格式
arrow_args = dict(arrowstyle = '<-')
 # 定义箭头格式
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
```

```
    '''
    函数功能:
            绘制节点
    参数说明:
            nodeTxt__节点名
            centerPt__文本位置
            parentPt__标注的箭头位置
            nodeType__节点格式
    '''
    createPlot.ax1.annotate(nodeTxt,
                    # 文本内容
                            xy = parentPt, xycoords = 'axes
fraction',          # 注释的起始位置，坐标系
                            xytext = centerPt, textcoords =
'axes fraction',    # 文本的起始位置
                            va = 'center', ha = 'center',
                    # 水平对齐，垂直对齐
                            bbox = nodeType,
                     # 节点格式
                            arrowprops = arrow_args)
                     # 箭头格式
```

- 边的标注

```
#标注有向边
def plotMidText(cntrPt, parentPt, txtString):
    '''
    函数功能:
            标注有向边内容
    参数说明:
            cntrpt、parentPt__计算标注位置
            txtString__标注内容
    '''
    xMid = (parentPt[0] - cntrPt[0])/ 2.0 + cntrPt[0]
   # 计算文本位置的横坐标
    yMid = (parentPt[1] - cntrPt[1])/2.0 + cntrPt[1]
  # 计算文本位置的纵坐标
    createPlot.ax1.text(xMid,
   # 文本位置的横坐标
                        yMid,
   # 文本位置的纵坐标
                        txtString)
  # 标注内容
```

- 绘制树

```
def plotTree(myTree, parentPt, nodeTxt):
    '''
    函数功能:
            绘制决策树
```

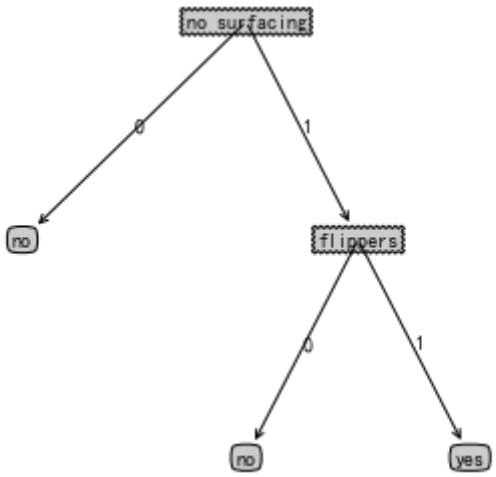
```
    函数参数：
            myTree__决策树
            parentPt__标注的内容
            nodeTxt__节点名称
    '''
    numLeafs = getNumLeafs(myTree)
 # 获取决策树叶结点数目，决定了树的宽度
    depth = getTreeDepth(myTree)
 # 获取决策树层数，决定了树的高度
    firstStr = next(iter(myTree))
  # 获得决策树第一个节点
    cntrPt = (plotTree.xOff+
(1.0+float(numLeafs))/2.0/plotTree.totalW,plotTree.yOff)#确
定中心位置
    plotMidText(cntrPt, parentPt, nodeTxt)
 # 标注有向边内容
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
 # 绘制节点
    secondDict = myTree[firstStr]
  # 获取下一个字典
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD
  # y偏移值
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':
 # 该结点是否为字典
            plotTree(secondDict[key], cntrPt, str(key))
  # 如果是字典则不是叶结点，递归调用继续绘制
        else:
            plotTree.xOff = plotTree.xOff +
1.0/plotTree.totalW                          # x偏移值
            plotNode(secondDict[key], (plotTree.xOff,
plotTree.yOff), cntrPt, leafNode)  # 绘制节点
            plotMidText((plotTree.xOff, plotTree.yOff),
cntrPt, str(key))                # 标注有向边内容
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD
```

- 画图：

```
def createPlot(inTree):
    '''
    函数功能：绘制完整的决策树
    参数说明：
            inTree__决策树
    '''
    fig = plt.figure(1, facecolor='white')
     #创建画布
    fig.clf()
      #清空画布
    axprops = dict(xticks=[], yticks=[])
```

```
    createPlot.ax1 = plt.subplot(111, frameon=False,
**axprops) # 除去x、y轴
    plotTree.totalW = float(getNumLeafs(inTree))
     # 获取决策树叶结点数目
    plotTree.totalD = float(getTreeDepth(inTree))
      # 获取决策树深度
    plotTree.xOff = -0.5/plotTree.totalW
     # x偏移的初始值
    plotTree.yOff = 1.0
      # y偏移的初始值
    plotTree(inTree, (0.5,1.0), '')
      # 绘制决策树
    plt.show()
     #显示图像

# 测试函数
# labels = ['no surfacing', 'flippers', 'labels']
# tree = createTree(dataSet, labels)
createPlot(tree)
```

结果：



保存树

```
def storeTree(inputTree, filename):
    '''
    函数功能：
            将决策树保存在磁盘中
    函数参数：
            inputTree__决策树
            filename__文件名
    '''
    import pickle                       # 导入pickle模块
    # 按照书中这里写的'w'，将会报错write() argument must be str,not
bytes
    # 所以这里将改写为'wb'
```

```
    fw = open(filename, 'wb')          # 创建一个可以“写入”的文件
    pickle.dump(inputTree, fw)         # pickle的dump函数将决策树写入文件
中
    fw.close()                         # 写完成后关闭文件
def gradTree(filename):
    '''
    函数功能:
            将树从磁盘中取出
    函数参数:
            filename__文件名
    '''
    import pickle                       # 导入pickle模块
    fr = open(filename, 'rb')           # 使用'rb'读出数据
    return pickle.load(fr)

# 函数测试
# labels = ['no surfacing', 'flippers', 'labels']
# tree = createTree(dataSet, labels)
storeTree(tree, 'classifer.json')
#gradTree('classifer.json')
```

测试隐形演讲类型

- 数据集
- 代码:

```
#查看决策树代码
from math import log
import matplotlib.pyplot as plt
def calcShannonEnt(dataSet):
    # 求list的长度，表示计算参与训练的数据量
    numEntries = len(dataSet)
    # 计算分类标签label出现的次数
    labelCounts = {}
    # the the number of unique elements and their
occurrence
    for featVec in dataSet:
        # 将当前实例的标签存储，即每一行数据的最后一个数据代表的是标
签
        currentLabel = featVec[-1]
        # 为所有可能的分类创建字典，如果当前的键值不存在，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。
        if currentLabel not in labelCounts.keys():
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1

    # 对于 label 标签的占比，求出 label 标签的香农熵
    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
        # 计算香农熵，以 2 为底求对数
        shannonEnt -= prob * log(prob, 2)
    return shannonEnt
#将某一个特征直接删除
def splitDataSet(dataSet, index, value):
    """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
    Args:
        dataSet 数据集                    待划分的数据集
        index 表示每一行的index列          划分数据集的特征
        value 表示index列对应的value值    需要返回的特征的值。
    Returns:
        index列为value的数据集【该数据集需要排除index列】
    """
    retDataSet = []
    for featVec in dataSet:
        # index列为value的数据集【该数据集需要排除index列】
        # 判断index列的值是否为value
        if featVec[index] == value:
            # chop out index used for splitting
            # [:index]表示前index行，即若 index 为2，就是取
featVec 的前 index 行
            reducedFeatVec = featVec[:index]
            '''
            请百度查询一下： extend和append的区别
            music_media.append(object) 向列表中添加一个对象
object
            music_media.extend(sequence) 把一个序列seq的内容
添加到列表中 (跟 += 在list运用类似， music_media += sequence)
            1、使用append的时候，是将object看作一个对象，整体打包
添加到music_media对象中。
            2、使用extend的时候，是将sequence看作一个序列，将这个
序列和music_media序列合并，并放在其后面。
            music_media = []
            music_media.extend([1,2,3])
            print music_media
            #结果:
            #[1, 2, 3]

            music_media.append([4,5,6])
            print music_media
            #结果:
            #[1, 2, 3, [4, 5, 6]]

            music_media.extend([7,8,9])
            print music_media
            #结果:
            #[1, 2, 3, [4, 5, 6], 7, 8, 9]
            '''
```

```
            reducedFeatVec.extend(featVec[index + 1:])
            # [index+1:]表示从跳过 index 的 index+1行，取接下
来的数据
            # 收集结果值 index列为value的行【该行需要排除index
列】
            retDataSet.append(reducedFeatVec)
    return retDataSet
#选择最好的数据集划分方式
def chooseBestFeatureToSplit(dataSet):
    """chooseBestFeatureToSplit(选择最好的特征)

    Args:
        dataSet 数据集
    Returns:
        bestFeature 最优的特征列
    """
    # 求第一行有多少列的 Feature, 最后一列是label列嘛
    numFeatures = len(dataSet[0]) - 1
    # 数据集的原始信息熵
    baseEntropy = calcShannonEnt(dataSet)
    # 最优的信息增益值, 和最优的Featurn编号
    bestInfoGain, bestFeature = 0.0, -1
    # iterate over all the features
    for i in range(numFeatures):
        # create a list of all the examples of this
feature
        # 获取对应的feature下的所有数据
        featList = [example[i] for example in dataSet]
        # get a set of unique values
        # 获取剔重后的集合，使用set对list数据进行去重
        uniqueVals = set(featList)
        # 创建一个临时的信息熵
        newEntropy = 0.0
        # 遍历某一列的value集合，计算该列的信息熵
        # 遍历当前特征中的所有唯一属性值，对每个唯一属性值划分一次数
据集，计算数据集的新熵值，并对所有唯一特征值得到的熵求和。
        for value in uniqueVals:
            subDataSet = splitDataSet(dataSet, i, value)
            # 计算概率
            prob = len(subDataSet)/float(len(dataSet))
            # 计算信息熵
            newEntropy += prob *
calcShannonEnt(subDataSet)
        # gain[信息增益]: 划分数据集前后的信息变化， 获取信息熵最大
的值
        # 信息增益是熵的减少或者是数据无序度的减少。最后，比较所有特
征中的信息增益，返回最好特征划分的索引值。
        infoGain = baseEntropy - newEntropy
        print('infoGain=', infoGain, 'bestFeature=', i,
baseEntropy, newEntropy)
        if (infoGain > bestInfoGain):
            bestInfoGain = infoGain
```

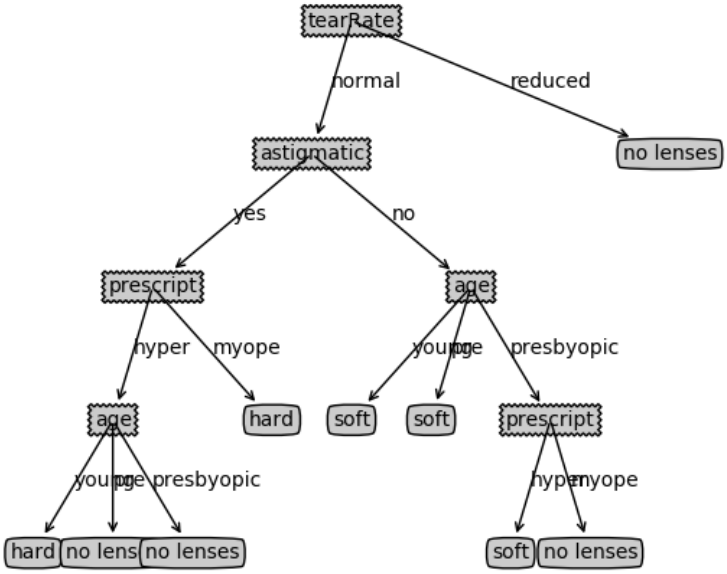
```
            bestFeature = i
    return bestFeature
# 类似于KNN中的投票机制
def majorityCnt(classList):
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
        else:
            classCount[vote] += 1
        sortedClassCount =
sorted(classCount.item(),key=lambda x:x[1],reverse = True)
        return sortedClassCount[0][0]
def createTree(dataSet, labels):
    classList = [example[-1] for example in dataSet]
    # 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。
    # count() 函数是统计括号中的值在list中出现的次数
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    # 如果数据集只有1列，那么最初出现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)

    # 选择最优的列，得到最优列对应的label含义
    bestFeat = chooseBestFeatureToSplit(dataSet)
    # 获取label的名称
    #print(bestFeat)
    #print(labels)
    bestFeatLabel = labels[bestFeat]
    # 初始化myTree
    myTree = {bestFeatLabel: {}}
    # 注：labels列表是可变对象，在PYTHON函数中作为参数时传址引用，
能够被全局修改
    # 所以这行代码导致函数外的同名变量被删除了元素，造成例句无法执
行，提示'no surfacing' is not in list
    del(labels[bestFeat])
    # 取出最优列，然后它的branch做分类
    print(labels)
    featValues = [example[bestFeat] for example in
dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        # 求出剩余的标签label
        subLabels = labels[:]
        # 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(),字典里面还有小字典
        #print(dataSet)
```

```
        myTree[bestFeatLabel][value] =
createTree(splitDataSet(dataSet, bestFeat, value),
subLabels)
        print(myTree)
        # print 'myTree', value, myTree
    return myTree
def createDataSet():
    dataSet = [[1, 1, 'yes'],
               [1, 1, 'yes'],
               [1, 0, 'no'],
               [0, 1, 'no'],
               [0, 1, 'no']]
    labels = ['no surfacing', 'flippers']
    return dataSet, labels
myDat,labels = createDataSet()
labels_copy = labels[:]
mytree = createTree(myDat,labels_copy)
# print(mytree)
# print(myDat)
#tree = createTree(myDat, labels)
print(labels)
tree = createTree(myDat,labels)
fr = open(r'C:\Users\admin\Desktop\machine-
learning\ai\DecisionTree\3.DecisionTree\lenses.txt')
lenses = [inst.strip().split('\t') for inst in
fr.readlines()]
lensesLable = ['age', 'prescript', 'astigmatic',
'tearRate']
lensesTree = createTree(lenses, lensesLable)
###############################################################
###
#获得叶子节点数目
def getNumLeafs(myTree):
    '''
    函数功能:
            递归计算叶子节点数目
    函数参数:
            字典形式的决策树
    函数返回:
            叶子节点数目
    '''
    numLeafs = 0
# 初始化叶子节点的数目
    print(list(myTree.keys()))
    firstStr = list(myTree.keys())[0]
 # 获取决策树的第一个节点
    secondDict = myTree[firstStr]
 # 获取决策树的第二个节点
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':
# 若该节点为字典形式
```

```
            numLeafs += getNumLeafs(secondDict[key])
# 若为字典，则递归计算新分支叶节点数
        else:
            numLeafs += 1
 # 若不是字典，则此节点为叶子节点
    return numLeafs
 # 返回叶子节点数目

# 函数测试
# labels = ['no surfacing', 'flippers', 'labels']
# labels_copy2 = labels[:]
# print(myDat)
# print(labels)
# tree = createTree(myDat,labels_copy2)
#得到树的深度
def getTreeDepth(myTree):
    '''
    函数功能：
            递归计算决策树的深度
    函数参数：
            myTree__字典形式的决策树
    函数返回：
            决策树的最大深度
    '''
    maxDepth = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:
            thisDepth = 1
        if thisDepth > maxDepth:
            maxDepth = thisDepth
    return maxDepth
#使用文本标注
decisionNode = dict(boxstyle = 'sawtooth', fc = '0.8')
 # 设置中间节点的格式
leafNode = dict(boxstyle = 'round4', fc = '0.8')
 # 设置叶子节点的格式
arrow_args = dict(arrowstyle = '<-')
 # 定义箭头格式
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    '''
    函数功能：
            绘制节点
    参数说明：
            nodeTxt__节点名
            centerPt__文本位置
            parentPt__标注的箭头位置
            nodeType__节点格式
    '''
```

```
    createPlot.ax1.annotate(nodeTxt,
                    # 文本内容
                            xy = parentPt, xycoords = 'axes
fraction',          # 注释的起始位置，坐标系
                            xytext = centerPt, textcoords =
'axes fraction',    # 文本的起始位置
                            va = 'center', ha = 'center',
                    # 水平对齐，垂直对齐
                            bbox = nodeType,
                     # 节点格式
                            arrowprops = arrow_args)
                     # 箭头格式
#标注有向边
def plotMidText(cntrPt, parentPt, txtString):
    '''
    函数功能：
            标注有向边内容
    参数说明：
            cntrpt、parentPt__计算标注位置
            txtString__标注内容
    '''
    xMid = (parentPt[0] - cntrPt[0])/ 2.0 + cntrPt[0]
   # 计算文本位置的横坐标
    yMid = (parentPt[1] - cntrPt[1])/2.0 + cntrPt[1]
  # 计算文本位置的纵坐标
    createPlot.ax1.text(xMid,
   # 文本位置的横坐标
                        yMid,
   # 文本位置的纵坐标
                        txtString)
  # 标注内容
def plotTree(myTree, parentPt, nodeTxt):
    '''
    函数功能：
            绘制决策树
    函数参数：
            myTree__决策树
            parentPt__标注的内容
            nodeTxt__节点名称
    '''
    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，决定了树的高度
    firstStr = next(iter(myTree))
  # 获得决策树第一个节点
    cntrPt = (plotTree.xOff+
(1.0+float(numLeafs))/2.0/plotTree.totalW,plotTree.yOff)#确
定中心位置
    plotMidText(cntrPt, parentPt, nodeTxt)
 # 标注有向边内容
```

```
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
 # 绘制节点
    secondDict = myTree[firstStr]
  # 获取下一个字典
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD
  # y偏移值
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':
 # 该结点是否为字典
            plotTree(secondDict[key], cntrPt, str(key))
  # 如果是字典则不是叶结点，递归调用继续绘制
        else:
            plotTree.xOff = plotTree.xOff +
1.0/plotTree.totalW                          # x偏移值
            plotNode(secondDict[key], (plotTree.xOff,
plotTree.yOff), cntrPt, leafNode)  # 绘制节点,y已经偏移
            plotMidText((plotTree.xOff, plotTree.yOff),
cntrPt, str(key))                # 标注有向边内容
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD
def createPlot(inTree):
    '''
    函数功能：绘制完整的决策树
    参数说明：
            inTree__决策树
    '''
    fig = plt.figure(1, facecolor='white')
     #创建画布
    fig.clf()
      #清空画布
    axprops = dict(xticks=[], yticks=[])
    createPlot.ax1 = plt.subplot(111, frameon=False,
**axprops) # 除去x、y轴
    plotTree.totalW = float(getNumLeafs(inTree))
     # 获取决策树叶结点数目
    plotTree.totalD = float(getTreeDepth(inTree))
      # 获取决策树深度
    plotTree.xOff = -0.5/plotTree.totalW
     # x偏移的初始值
    plotTree.yOff = 1.0
      # y偏移的初始值
    plotTree(inTree, (0.5,1.0), '')
      # 绘制决策树,父节点(0.5,1.0)
    plt.show()
     #显示图像

# 测试函数
# labels = ['no surfacing', 'flippers', 'labels']
# tree = createTree(dataSet, labels)
# createPlot(tree)
createPlot(lensesTree)
```



随机森林 random forest

- 数据集地址

概述：随机森林是指多棵树对样本进行训练并且预测的一种分类器，决策树相当于大师，通过自己在数据集中学习到的只是用于新数据的分类，三个臭皮匠，顶个诸葛亮

原理：

- 数据的随机化：使得随机森林中的决策树更普遍化一点，适合更多的场景。

  （有放回的准确率在：70% 以上， 无放回的准确率在：60% 以上）

  a. 采取有放回的抽样方式 构造子数据集，保证不同子集之间的数量级一样（不同子集 / 同一子集 之间的元素可以重复）
  b. 利用子数据集来构建子决策树，将这个数据放到每个子决策树中，每个子决策树输出一个结果。
  c. 然后统计子决策树的投票结果，得到最终的分类 就是 随机森林的输出结果。
  d. 如下图，假设随机森林中有3棵子决策树，2棵子树的分类结果是A类，1棵子树的分类结果是B类，那么随机森林的分类结果就是A类
- 待选择特征的随机化:

  a. 子树从所有的待选特征中随机选取一定的特征。
  b. 在选取的特征中选取最优的特征。

下图中，蓝色的方块代表所有可以被选择的特征，也就是目前的待选特征；黄色的方块是分裂特征。
左边是一棵决策树的特征选取过程，通过在待选特征中选取最优的分裂特征（别忘了前文提到的ID3算法，C4.5算法，CART算法等等），完成分裂。右边是一个随机森林中的子树的特征选取过程。

随机森林中的决策树彼此不同，提升系统的多样性，提升分类性能

开发：

```
收集数据：任何方法
准备数据：转换样本集
分析数据：任何方法
训练算法：通过数据随机化和特征随机化，进行多实例的分类评估
测试算法：计算错误率
使用算法：输入样本数据，然后运行 随机森林 算法判断输入数据分类属于哪个分类，最后对计算出的分类执行后续处理
```

- 优缺点：

```
优点：几乎不需要输入准备、可实现隐式特征选择、训练速度非常快、其他模型很难超越、很难建立一个糟糕的随机森林模型、大量优秀、免费以及开源的实现。
缺点：劣势在于模型大小、是个很难去解释的黑盒子。
适用数据范围：数值型和标称型
```

- 导入样本集:

```
def loadDataSet(filename:str):
    dataset = []
    with open(filename,'r') as f:
        for line in f.readlines():
            if not line:
                continue
            lineArr = []
            # le = len(line.split(','))
            for featrue in line.split(','):
                #strip()返回左右去除空格
                str_f = featrue.strip()
                # if str_f.isdigit():# 判断是否可以转化为数字
                #     # 将这一行数据的某一列可以转化为数字的转化为float
                #     lineArr.append(float(str_f))
                # else:
                #     #添加label
                #     lineArr.append(str_f)
                try:
                    lineArr.append(float(str_f))
                except:
                    lineArr.append(str_f)
            dataset.append(lineArr)
    return dataset
```

- 交叉验证,将样本集合划分成几个集合

```
#交叉验证,给数据集合划分不是给数据划分
```

```
def cross_validation_split(dataset, n_folds):
    """cross_validation_split(将数据集进行抽重抽样 n_folds
份，数据可以重复重复抽取，每一次list的元素是无重复的)
    Args:
        dataset      原始数据集
        n_folds      数据集dataset分成n_flods份
    Returns:
        dataset_split    list集合，存放的是：将数据集进行抽重抽样
n_folds 份，数据可以重复重复抽取，每一次list的元素是无重复的
    """
    dataset_split = list()
    dataset_copy = list(dataset)        # 复制一份 dataset,防
止 dataset 的内容改变
    fold_size = len(dataset) / n_folds
    for i in range(n_folds):
        fold = list()                   # 每次循环 fold 清零，
防止重复导入 dataset_split
        while len(fold) < fold_size:    # 这里不能用 if，if 只
是在第一次判断时起作用，while 执行循环，直到条件不成立
            # 有放回的随机采样，有一些样本被重复采样，从而在训练集中
多次出现，有的则从未在训练集中出现，此则自助采样法。从而保证每棵决策树
训练集的差异性
            index = randrange(len(dataset_copy))
            # 将对应索引 index 的内容从 dataset_copy 中导出，并
将该内容从 dataset_copy 中删除。
            # pop() 函数用于移除列表中的一个元素（默认最后一个元
素），并且返回该元素的值。
            # fold.append(dataset_copy.pop(index))  # 无放
回的方式
            fold.append(dataset_copy[index])  # 有放回的方式
        dataset_split.append(fold)
    # 由dataset分割出的n_folds个数据构成的列表，为了用于交叉验证
    return dataset_split
```

- 通过指定值进行分割数据,可以理解为树的两个孩子节点

```
# Split a dataset based on an attribute and an attribute
value # 根据特征和特征值分割数据集
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if float(row[index]) < float(value):
            left.append(row)
        else:
            right.append(row)
    return left, right
```

- 基尼不纯系数的计算

```
# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):    # 个人理解：计算代价，分类越准确，则 gini 越小
    gini = 0.0
    for class_value in class_values:     # class_values = [0, 1]就是label的值
        for group in groups:             # groups = (left, right)
            size = len(group)
            if size == 0:
                continue
            proportion = [row[-1] for row in group].count(class_value) / float(size)
            gini += (proportion * (1.0 - proportion))    # 个人理解：计算代价，分类越准确，则 gini 越小
    return gini
```

- 分割数据的最优特征

```
# 找出分割数据集的最优特征，得到最优的特征 index，特征值 row[index]，以及分割完的数据 groups（left, right）
def get_split(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))
 # class_values =[0, 1]
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1)  # 往 features 添加 n_features 个特征（ n_feature 等于特征数的根号），特征索引从 dataset 中随机取
        if index not in features:
            features.append(index)
    for index in features:                      # 在 n_features 个特征中选出最优的特征索引，并没有遍历所有特征，从而保证了每课决策树的差异性
        for row in dataset:
            groups = test_split(index, row[index], dataset)  # groups=(left, right), row[index] 遍历每一行 index 索引下的特征值作为分类值 value，找出最优的分类特征和特征值
            gini = gini_index(groups, class_values)
            # 左右两边的数量越一样，说明数据区分度不高，gini系数越大
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups  # 最后得到最优的分类特征 b_index,分类特征值 b_value,分类结果 b_groups。b_value 为分错的代价成本
    # print(b_score)
    return {'index': b_index, 'value': b_value, 'groups': b_groups}
```

- 统计结果中最好的特征

```
# Create a terminal node value # 输出group中出现次数较多的标签
def to_terminal(group):
    outcomes = [row[-1] for row in group]            #
max() 函数中，当 key 参数不为空时，就以 key 的函数对象为判断的标准
    return max(set(outcomes), key=outcomes.count)   # 输出
group 中出现次数较多的标签
```

- 创建一个决策树

```
# Create child splits for a node or make terminal  # 创建子
分割器，递归分类，直到分类结束
def split(node, max_depth, min_size, n_features, depth):
 # max_depth = 10, min_size = 1, n_features =
int(sqrt((dataset[0])-1))
    left, right = node['groups']
    del(node['groups'])
# check for a no split
    if not left or not right:
        node['left'] = node['right'] = to_terminal(left +
right)
        return
# check for max depth
    if depth >= max_depth:   # max_depth=10 表示递归十次，若
分类还未结束，则选取数据中分类标签较多的作为结果，使分类提前结束，防止
过拟合
        node['left'], node['right'] = to_terminal(left),
to_terminal(right)
        return
# process left child
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left, n_features)  #
node['left']是一个字典，形式为{'index':b_index,
'value':b_value, 'groups':b_groups}，所以node是一个多层字典
        split(node['left'], max_depth, min_size,
n_features, depth+1)  # 递归，depth+1计算递归层数
# process right child
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
        node['right'] = get_split(right, n_features)
        split(node['right'], max_depth, min_size,
n_features, depth+1)


# Build a decision tree
def build_tree(train, max_depth, min_size, n_features):
    """build_tree(创建一个决策树)
    Args:
```

```
        train           训练数据集
        max_depth       决策树深度不能太深，不然容易导致过拟合
        min_size        叶子节点的大小
        n_features      选取的特征的个数
    Returns:
        root            返回决策树
    """

    # 返回最优列和相关的信息
    root = get_split(train, n_features)

    # 对左右2边的数据 进行递归的调用，由于最优特征使用过，所以在后面进行使用的时候，就没有意义了
    # 例如： 性别-男女，对男使用这一特征就没任何意义了
    split(root, max_depth, min_size, n_features, 1)
    return root
```

- 通过决策树来进行预测

```
# Make a prediction with a decision tree
def predict(node, row):   # 预测模型分类结果
    if float(row[node['index']]) < float(node['value']):
        if isinstance(node['left'], dict):       # isinstance 是 Python 中的一个内建函数。是用来判断一个对象是否是一个已知的类型。
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']
```

- 随机森林预测

```
# Make a prediction with a list of bagged trees
def bagging_predict(trees, row):
    """bagging_predict(bagging预测)
    Args:
        trees           决策树的集合
        row             测试数据集的每一行数据
    Returns:
        返回随机森林中，决策树结果出现次数做大的
    """

    # 使用多个决策树trees对测试集test的第row行进行预测，再使用简单投票法判断出该行所属分类
    predictions = [predict(tree, row) for tree in trees]
    return max(set(predictions), key=predictions.count)
```

- 数据随机样本

```
# 创建数据集合中的随机子样本
# Create a random subsample from the dataset with
replacement
def subsample(dataset, ratio):   # 创建数据集的随机子样本
    """random_forest(评估算法性能，返回模型得分)
    Args:
        dataset          训练数据集
        ratio            训练数据集的样本比例
    Returns:
        sample           随机抽样的训练样本
    """

    sample = list()
    # 训练样本的按比例抽样。
    # round() 方法返回浮点数x的四舍五入值。
    n_sample = round(len(dataset) * ratio)
    while len(sample) < n_sample:
        # 有放回的随机采样，有一些样本被重复采样，从而在训练集中多次
出现，有的则从未在训练集中出现，此则自助采样法。从而保证每棵决策树训练
集的差异性
        index = randrange(len(dataset))
        sample.append(dataset[index])
    return sample
```

- 随机森林预测算法,对所有测试集合

```
# Random Forest Algorithm
def random_forest(train, test, max_depth, min_size,
sample_size, n_trees, n_features):
    """random_forest(评估算法性能，返回模型得分)
    Args:
        train            训练数据集
        test             测试数据集
        max_depth        决策树深度不能太深，不然容易导致过拟合
        min_size         叶子节点的大小
        sample_size      训练数据集的样本比例
        n_trees          决策树的个数
        n_features       选取的特征的个数
    Returns:
        predictions      每一行的预测结果，bagging 预测最后的分类
结果
    """

    trees = list()
    # n_trees 表示决策树的数量
    for i in range(n_trees):
```

```
        # 随机抽样的训练样本， 随机采样保证了每棵决策树训练集的差异
性
        sample = subsample(train, sample_size)
        # 创建一个决策树
        tree = build_tree(sample, max_depth, min_size,
n_features)
        trees.append(tree)

    # 每一行的预测结果，bagging 预测最后的分类结果
    predictions = [bagging_predict(trees, row) for row in
test]
    return predictions
```

- 通过预测值和真实值求出准确率

```
# Calculate accuracy percentage
def accuracy_metric(actual, predicted):  # 导入实际值和预测
值，计算精确度
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0
```

- rerank函数:对一系列交叉验证的结果

```
# 评估算法性能，返回模型得分
def evaluate_algorithm(dataset, algorithm, n_folds,
*args):
    """evaluate_algorithm(评估算法性能，返回模型得分)
    Args:
        dataset     原始数据集
        algorithm   使用的算法
        n_folds     数据的份数
        *args       其他的参数
    Returns:
        scores      模型得分
    """

    # 将数据集进行抽重抽样 n_folds 份，数据可以重复重复抽取，每一次
list 的元素是无重复的
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    # 每次循环从 folds 从取出一个 fold 作为测试集，其余作为训练集，
遍历整个 folds ，实现交叉验证
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        # 将多个 fold 列表组合成一个 train_set 列表, 类似 union
all
        """
```

```
        In [20]: l1=[[1, 2, 'a'], [11, 22, 'b']]
        In [21]: l2=[[3, 4, 'c'], [33, 44, 'd']]
        In [22]: l=[]
        In [23]: l.append(l1)
        In [24]: l.append(l2)
        In [25]: l
        Out[25]: [[[1, 2, 'a'], [11, 22, 'b']], [[3, 4,
'c'], [33, 44, 'd']]]
        In [26]: sum(l, [])
        Out[26]: [[1, 2, 'a'], [11, 22, 'b'], [3, 4, 'c'],
[33, 44, 'd']]
        """
        train_set = sum(train_set, [])
        test_set = list()
        # fold 表示从原始数据集 dataset 提取出来的测试集
        for row in fold:
            row_copy = list(row)
            row_copy[-1] = None
            test_set.append(row_copy)
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]

        # 计算随机森林的预测结果的正确率
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores
```

- 总体代码

```
# 随机森林 randomforest
# 收集数据：提供的文本文件
# 准备数据：转换样本集
# 分析数据：手工检查数据
# 训练算法：在数据上，利用 random_forest() 函数进行优化评估，返回模型的综合分类结果
# 测试算法：在采用自定义 n_folds 份随机重抽样 进行测试评估，得出综合的预测评分
# 使用算法：若你感兴趣可以构建完整的应用程序，从案例进行封装，也可以参考我们的代码
from random import seed,randrange,random
def loadDataSet(filename:str):
    dataset = []
    with open(filename,'r') as f:
        for line in f.readlines():
            if not line:
                continue
            lineArr = []
            # le = len(line.split(','))
            for featrue in line.split(','):
                #strip()返回左右去除空格
                str_f = featrue.strip()
                # if str_f.isdigit():# 判断是否可以转化为数字
```

```
                #     # 将这一行数据的某一列可以转化为数字的转化为
float
                #     lineArr.append(float(str_f))
                # else:
                #     #添加label
                #     lineArr.append(str_f)
                try:
                    lineArr.append(float(str_f))
                except:
                    lineArr.append(str_f)
            dataset.append(lineArr)
    return dataset
#交叉验证,给数据集合划分不是给数据划分
def cross_validation_split(dataset, n_folds):
    """cross_validation_split(将数据集进行抽重抽样 n_folds
份，数据可以重复重复抽取，每一次list的元素是无重复的)
    Args:
        dataset     原始数据集
        n_folds     数据集dataset分成n_flods份
    Returns:
        dataset_split    list集合，存放的是：将数据集进行抽重抽样
n_folds 份，数据可以重复重复抽取，每一次list的元素是无重复的
    """
    dataset_split = list()
    dataset_copy = list(dataset)        # 复制一份 dataset,防
止 dataset 的内容改变
    fold_size = len(dataset) / n_folds
    for i in range(n_folds):
        fold = list()                   # 每次循环 fold 清零，
防止重复导入 dataset_split
        while len(fold) < fold_size:    # 这里不能用 if，if 只
是在第一次判断时起作用，while 执行循环，直到条件不成立
            # 有放回的随机采样，有一些样本被重复采样，从而在训练集中
多次出现，有的则从未在训练集中出现，此则自助采样法。从而保证每棵决策树
训练集的差异性
            index = randrange(len(dataset_copy))
            # 将对应索引 index 的内容从 dataset_copy 中导出，并
将该内容从 dataset_copy 中删除。
            # pop() 函数用于移除列表中的一个元素（默认最后一个元
素），并且返回该元素的值。
            # fold.append(dataset_copy.pop(index))  # 无放
回的方式
            fold.append(dataset_copy[index])  # 有放回的方式
        dataset_split.append(fold)
    # 由dataset分割出的n_folds个数据构成的列表，为了用于交叉验证
    return dataset_split


# Split a dataset based on an attribute and an attribute
value # 根据特征和特征值分割数据集
def test_split(index, value, dataset):
    left, right = list(), list()
```

```
    for row in dataset:
        if float(row[index]) < float(value):
            left.append(row)
        else:
            right.append(row)
    return left, right


# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):    # 个人理解：计算代价，分类越准确，则 gini 越小
    gini = 0.0
    for class_value in class_values:     # class_values = [0, 1]就是label的值
        for group in groups:             # groups = (left, right)
            size = len(group)
            if size == 0:
                continue
            proportion = [row[-1] for row in group].count(class_value) / float(size)
            gini += (proportion * (1.0 - proportion))    # 个人理解：计算代价，分类越准确，则 gini 越小
    return gini


# 找出分割数据集的最优特征，得到最优的特征 index，特征值 row[index]，以及分割完的数据 groups（left, right）
def get_split(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))
 # class_values =[0, 1]
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1)  # 往 features 添加 n_features 个特征（ n_feature 等于特征数的根号），特征索引从 dataset 中随机取
        if index not in features:
            features.append(index)
    for index in features:                      # 在 n_features 个特征中选出最优的特征索引，并没有遍历所有特征，从而保证了每课决策树的差异性
        for row in dataset:
            groups = test_split(index, row[index], dataset)  # groups=(left, right), row[index] 遍历每一行 index 索引下的特征值作为分类值 value，找出最优的分类特征和特征值
            gini = gini_index(groups, class_values)
            # 左右两边的数量越一样，说明数据区分度不高，gini系数越大
            if gini < b_score:
```

```
                b_index, b_value, b_score, b_groups =
index, row[index], gini, groups  # 最后得到最优的分类特征
b_index,分类特征值 b_value,分类结果 b_groups。b_value 为分错的
代价成本
    # print(b_score)
    return {'index': b_index, 'value': b_value, 'groups':
b_groups}


# Create a terminal node value # 输出group中出现次数较多的标签
def to_terminal(group):
    outcomes = [row[-1] for row in group]            #
max() 函数中，当 key 参数不为空时，就以 key 的函数对象为判断的标准
    return max(set(outcomes), key=outcomes.count)   # 输出
group 中出现次数较多的标签


# Create child splits for a node or make terminal  # 创建子
分割器，递归分类，直到分类结束
def split(node, max_depth, min_size, n_features, depth):
 # max_depth = 10, min_size = 1, n_features =
int(sqrt((dataset[0])-1))
    left, right = node['groups']
    del(node['groups'])
# check for a no split
    if not left or not right:
        node['left'] = node['right'] = to_terminal(left +
right)
        return
# check for max depth
    if depth >= max_depth:   # max_depth=10 表示递归十次，若
分类还未结束，则选取数据中分类标签较多的作为结果，使分类提前结束，防止
过拟合
        node['left'], node['right'] = to_terminal(left),
to_terminal(right)
        return
# process left child
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left, n_features)  #
node['left']是一个字典，形式为{'index':b_index,
'value':b_value, 'groups':b_groups}，所以node是一个多层字典
        split(node['left'], max_depth, min_size,
n_features, depth+1)  # 递归，depth+1计算递归层数
# process right child
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
        node['right'] = get_split(right, n_features)
        split(node['right'], max_depth, min_size,
n_features, depth+1)
```

```
# Build a decision tree
def build_tree(train, max_depth, min_size, n_features):
    """build_tree(创建一个决策树)
    Args:
        train            训练数据集
        max_depth        决策树深度不能太深，不然容易导致过拟合
        min_size         叶子节点的大小
        n_features       选取的特征的个数
    Returns:
        root             返回决策树
    """

    # 返回最优列和相关的信息
    root = get_split(train, n_features)

    # 对左右2边的数据 进行递归的调用，由于最优特征使用过，所以在后面进行使用的时候，就没有意义了
    # 例如： 性别-男女，对男使用这一特征就没任何意义了
    split(root, max_depth, min_size, n_features, 1)
    return root


# Make a prediction with a decision tree
def predict(node, row):   # 预测模型分类结果
    if float(row[node['index']]) < float(node['value']):
        if isinstance(node['left'], dict):       # isinstance 是 Python 中的一个内建函数。是用来判断一个对象是否是一个已知的类型。
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']


# Make a prediction with a list of bagged trees
def bagging_predict(trees, row):
    """bagging_predict(bagging预测)
    Args:
        trees            决策树的集合
        row              测试数据集的每一行数据
    Returns:
        返回随机森林中，决策树结果出现次数做大的
    """

    # 使用多个决策树trees对测试集test的第row行进行预测，再使用简单投票法判断出该行所属分类
```

```
    predictions = [predict(tree, row) for tree in trees]
    return max(set(predictions), key=predictions.count)

# 创建数据集合中的随机子样本
# Create a random subsample from the dataset with
replacement
def subsample(dataset, ratio):   # 创建数据集的随机子样本
    """random_forest(评估算法性能，返回模型得分)
    Args:
        dataset          训练数据集
        ratio            训练数据集的样本比例
    Returns:
        sample           随机抽样的训练样本
    """

    sample = list()
    # 训练样本的按比例抽样。
    # round() 方法返回浮点数x的四舍五入值。
    n_sample = round(len(dataset) * ratio)
    while len(sample) < n_sample:
        # 有放回的随机采样，有一些样本被重复采样，从而在训练集中多次
出现，有的则从未在训练集中出现，此则自助采样法。从而保证每棵决策树训练
集的差异性
        index = randrange(len(dataset))
        sample.append(dataset[index])
    return sample


# Random Forest Algorithm
def random_forest(train, test, max_depth, min_size,
sample_size, n_trees, n_features):
    """random_forest(评估算法性能，返回模型得分)
    Args:
        train            训练数据集
        test             测试数据集
        max_depth        决策树深度不能太深，不然容易导致过拟合
        min_size         叶子节点的大小
        sample_size      训练数据集的样本比例
        n_trees          决策树的个数
        n_features       选取的特征的个数
    Returns:
        predictions      每一行的预测结果，bagging 预测最后的分类
结果
    """

    trees = list()
    # n_trees 表示决策树的数量
    for i in range(n_trees):
        # 随机抽样的训练样本， 随机采样保证了每棵决策树训练集的差异
性
        sample = subsample(train, sample_size)
        # 创建一个决策树
```

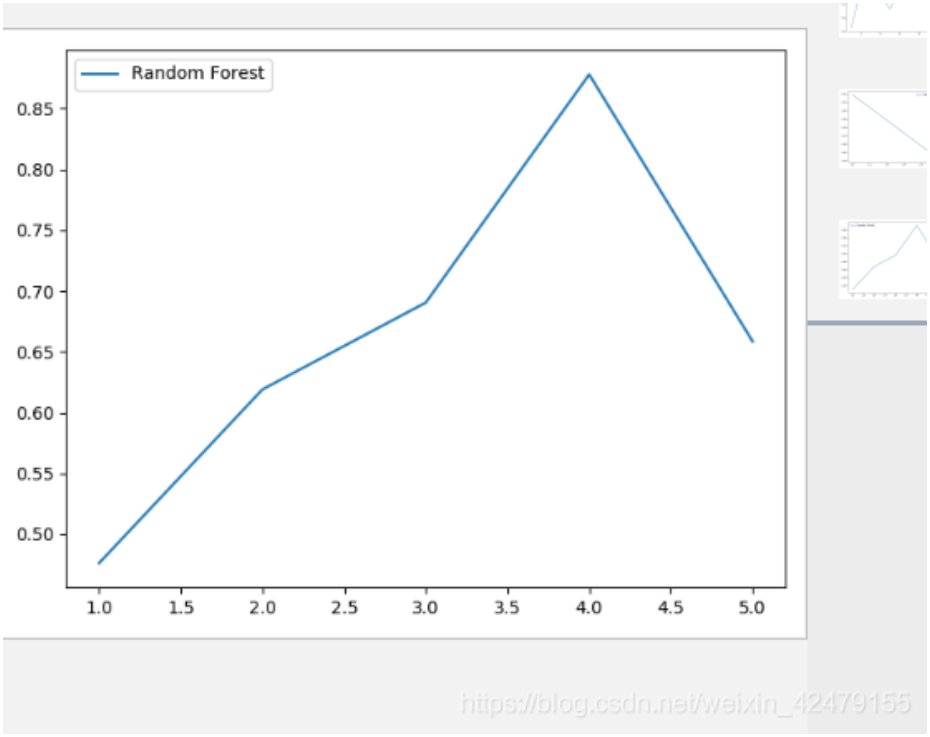
```
        tree = build_tree(sample, max_depth, min_size,
n_features)
        trees.append(tree)

    # 每一行的预测结果，bagging 预测最后的分类结果
    predictions = [bagging_predict(trees, row) for row in
test]
    return predictions


# Calculate accuracy percentage
def accuracy_metric(actual, predicted):  # 导入实际值和预测
值，计算精确度
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0


# 评估算法性能，返回模型得分
def evaluate_algorithm(dataset, algorithm, n_folds,
*args):
    """evaluate_algorithm(评估算法性能，返回模型得分)
    Args:
        dataset     原始数据集
        algorithm   使用的算法
        n_folds     数据的份数
        *args       其他的参数
    Returns:
        scores      模型得分
    """

    # 将数据集进行抽重抽样 n_folds 份，数据可以重复重复抽取，每一次
list 的元素是无重复的
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    # 每次循环从 folds 从取出一个 fold 作为测试集，其余作为训练集，
遍历整个 folds ，实现交叉验证
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        # 将多个 fold 列表组合成一个 train_set 列表, 类似 union
all
        """
        In [20]: l1=[[1, 2, 'a'], [11, 22, 'b']]
        In [21]: l2=[[3, 4, 'c'], [33, 44, 'd']]
        In [22]: l=[]
        In [23]: l.append(l1)
        In [24]: l.append(l2)
        In [25]: l
```

```
        Out[25]: [[[1, 2, 'a'], [11, 22, 'b']], [[3, 4,
'c'], [33, 44, 'd']]]
        In [26]: sum(l, [])
        Out[26]: [[1, 2, 'a'], [11, 22, 'b'], [3, 4, 'c'],
[33, 44, 'd']]
        """
        train_set = sum(train_set, [])
        test_set = list()
        # fold 表示从原始数据集 dataset 提取出来的测试集
        for row in fold:
            row_copy = list(row)
            row_copy[-1] = None
            test_set.append(row_copy)
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]

        # 计算随机森林的预测结果的正确率
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores


if __name__ == '__main__':

    # 加载数据
    dataset = loadDataSet('/home/ach/桌面/machine-
learning/ai/randomforest/7.RandomForest/sonar-all-
data.txt')
    # print(dataset)

    n_folds = 10         # 分成5份数据，进行交叉验证
    max_depth = 20      # 调参（自己修改） #决策树深度不能太深，不
然容易导致过拟合
    min_size = 1        # 决策树的叶子节点最少的元素数量
    sample_size = 1.0   # 做决策树时候的样本的比例
    # n_features = int((len(dataset[0])-1))
    n_features = 15      # 调参（自己修改） #准确性与多样性之间的
权衡
    for n_trees in [1, 10, 20]:  # 理论上树是越多越好
        scores = evaluate_algorithm(dataset,
random_forest, n_folds, max_depth, min_size, sample_size,
n_trees, n_features)
        # 每一次执行本文件时都能产生同一个随机数
        seed(1)
        print('random=', random())
        print('Trees: %d' % n_trees)
        print('Scores: %s' % scores)
        print('Mean Accuracy: %.3f%%' %
(sum(scores)/float(len(scores))))
```

SK 实现随机森林

```
# 导入需要的包
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import  RandomForestClassifier
from sklearn.model_selection import cross_val_score
def loadDataSet(filename:str):
    dataset = []
    datalabel = []
    with open(filename,'r') as f:
        for line in f.readlines():
            if not line:
                continue
            lineArr = []
            # le = len(line.split(','))
            for featrue in line.split(','):
                #strip()返回左右去除空格
                str_f = featrue.strip()
                # if str_f.isdigit():# 判断是否可以转化为数字
                #     # 将这一行数据的某一列可以转化为数字的转化为float
                #     lineArr.append(float(str_f))
                # else:
                #     #添加label
                #     # lineArr.append(str_f)
                #     datalabel.append(str_f)
                try:
                    lineArr.append(float(str_f))
                except:
                    datalabel.append(str_f)
            dataset.append(lineArr)
    return dataset,datalabel
if __name__ == '__main__':
    # 加载数据
    dataset,datalabel = loadDataSet('/home/ach/桌面/machine-
learning/ai/randomforest/7.RandomForest/sonar-all-data.txt')
    rfc =
RandomForestClassifier(random_state=1,n_estimators=11,oob_score=Tru
e,max_features=15)
    # print(np.mat(dataset).shape)
    # print(np.mat(datalabel).reshape())
    #
print(cross_val_score(rfc,np.mat(dataset),np.mat(datalabel).T,cv=10
).mean())
    score =
cross_val_score(rfc,np.mat(dataset),np.mat(datalabel).T,cv = 5)
    plt.plot(range(1,6),score,label = "Random Forest")
    plt.legend()
    plt.show()
```

- 运行结果



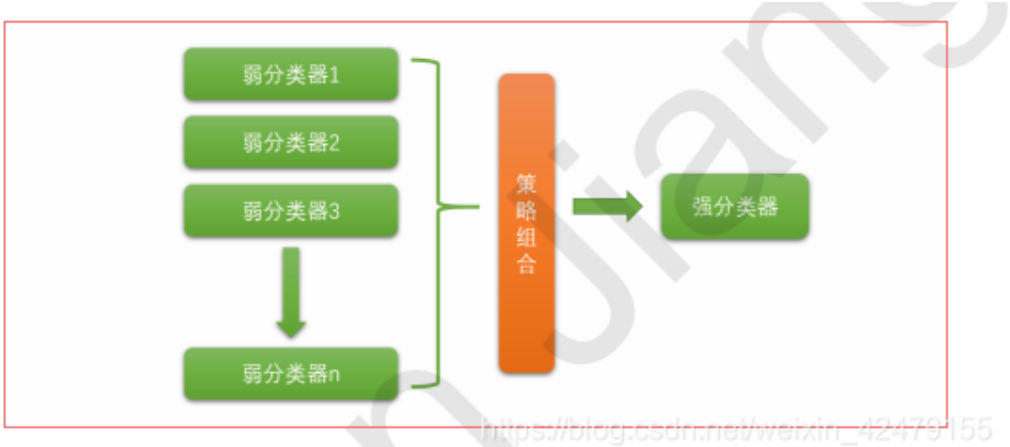
https://blog.csdn.net/weixin_42479155

Adaboost算法

集成学习概述

集成学习算法定义

- 集成学习(Ensemble learning)就是讲若干个弱分类器通过一定策略组合后产生一个强分类器。弱分类器(weak Classifier)指的就是那些分类准确率只比随机猜测好一点的分类器。而强分类器(strong Classifier)的分类准确率会高很多，这里的弱和强是相对的，弱分类器也叫做**基分类器**



https://blog.csdn.net/weixin_42479155

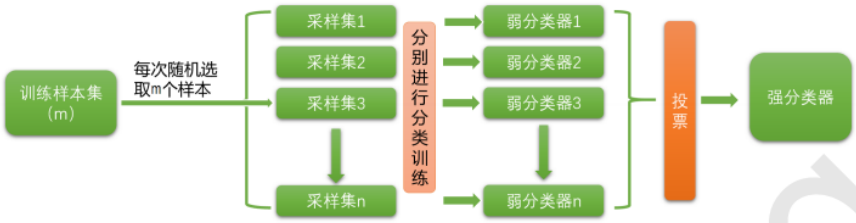
- 分类：

  bagging

boosting

bagging（装袋）

- bagging方法又叫做自举汇聚法(boostrap aggregating),是一种根据均匀概率分布从数据集中重复抽样(有放回)的技术，每个数据集和原始数据集大小相等，由于新数据集的每一个样本都是从原数据集合中有放回随机抽样出来的，所以每个数据集中可能有重复的值，而原始数据集中的某些样本可能根本就没有出现在新数据集中



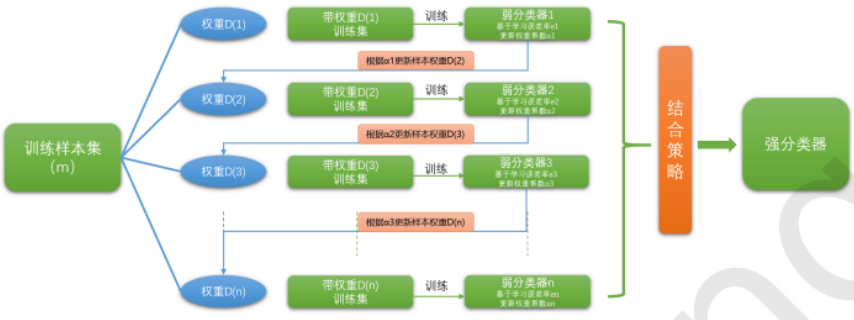
bagging图示

https://blog.csdn.net/weixin_42479155

- 有放回的随机抽样：自主采样法(Bootstap sampling),也就是说对于m个样本的原始数据集，每次随机选取一个样本放回采样集合中，然后这个样本重新放回，再进行下一次随机抽样，直到采样集合中样本数量达到m，这样一个采样集合就构建好了，重复过程，生成n个采样集合
- 将n个采样集合，分别进行训练，得到n个弱分类器，根据每个结果进行组合，得到强分类器
- 降低弱分类器的方差

boosting(提升)

- 迭代过程，用来自适应的改变训练样本的分布，使得弱分类器聚焦到那些很难分类的样本上，它的做法是给每一个训练样本赋予一个权重，在每一轮训练中自动调整权重



boosting图示

https://blog.csdn.net/weixin_42479155

boosting方法的代表算法有Adaboost、GBDT、XGBoost算法

- 组合策略

    a. 平均法：

对于数值类的回归预测问题，通常使用的结合策略是平均法，也就是说，对于若干个弱学习器的输出进行平均得到最终的预测输出。

假设我们最终得到的n个弱分类器为$\{h_1, h_2, \ldots, h_n\}$

最简单的平均是算术平均，也就是说最终预测是

$$H(x) = \frac{1}{n}\sum_{1}^{n} h_i(x)$$

如果每个弱分类器有一个权重w，则最终预测是

$$H(x) = \frac{1}{n}\sum_{1}^{n} w_i h_i(x)$$

$$s.t.\ \ w_i \geq 0, \sum_{1}^{n} w_i = 1$$

https://blog.csdn.net/weixin_42479155

b. 投票法：

对于分类问题的预测，我们通常使用的是投票法。假设我们的预测类别是$\{c_1, c_2, \ldots c_K\}$,对于任意一个预测样本x，我们的n个弱学习器的预测结果分别是$(h_1(x), h_2(x) \ldots h_n(x))$。

最简单的投票法是相对多数投票法，也就是我们常说的少数服从多数，也就是n个弱学习器的对样本x的预测结果中，数量最多的类别c_i为最终的分类类别。如果不止一个类别获得最高票，则随机选择一个做最终类别。

稍微复杂的投票法是绝对多数投票法，也就是我们常说的要票过半数。在相对多数投票法的基础上，不光要求获得最高票，还要求票过半数。否则会拒绝预测。

c. 学习法：

前两种方法都是对弱学习器的结果做平均或者投票，相对比较简单，但是可能学习误差较大，于是就有了学习法这种方法，对于学习法，代表方法是stacking，当使用stacking的结合策略时，我们不是对弱学习器的结果做简单的逻辑处理，而是再加上一层学习器，也就是说，我们将训练集弱学习器的学习结果作为输入，将训练集的输出作为输出，重新训练一个学习器来得到最终结果。

在这种情况下，我们将弱学习器称为初级学习器，将用于结合的学习器称为次级学习器。对于测试集，我们首先用初级学习器预测一次，得到次级学习器的输入样本，再用次级学习器预测一次，得到最终的预测结果。

Adaboost算法(自适应提升算法)

1. 计算样本权重

赋予训练集中每个样本一个权重，构成权重向量D，将权重向量D初始化相等值。

假设我们有n个样本的训练集：

$$\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

设定每个样本的权重都相等，则权重为$\frac{1}{n}$。

2. 计算错误率

在训练集上训练出一个弱分类器，并计算分类器的错误率：

$$\epsilon = \frac{分错的数量}{样本总数}$$

3. 计算弱分类器权重

为当前分类器赋予权重值alpha，则alpha计算公式为：

$$\alpha = \frac{1}{2} ln(\frac{1-\epsilon}{\epsilon})$$

4. 调整权重值

根据上一次训练结果，调整权重值（上一次分对的权重降低，分错的权重增加）

如果第i个样本被正确分类，则该样本权重更改为：

$$D_i^{(t+1)} = \frac{D_i^{(t)} e^{-\alpha}}{Sum(D)}$$

如果第i个样本被分错，则该样本权重更改为：

$$D_i^{(t+1)} = \frac{D_i^{(t)} e^{\alpha}}{Sum(D)}$$

https://blog.csdn.net/weixin_42479155

- 收集数据：可以使用任意方法
  准备数据：依赖于所使用的弱分类器类型，本章使用的是单层决策树，这种分类器可以处理任何数据类型。
  　当然也可以使用任意分类器作为弱分类器，第2章到第6章中的任一分类器都可以充当弱分类器。
  　作为弱分类器，简单分类器的效果更好。
  分析数据：可以使用任意方法。
  训练算法：AdaBoost 的大部分时间都用在训练上，分类器将多次在同一数据集上训练弱分类器。
  测试算法：计算分类的错误率。
  使用算法：通SVM一样，AdaBoost 预测两个类别中的一个。如果想把它应用到多个类别的场景，那么就要像多类 SVM 中的做法一样对 AdaBoost
- 优点：泛化（由具体的、个别的扩大为一般的）错误率低，易编码，可以应用在大部分分类器上，无参数调节。

  缺点：对离群点敏感。

  适用数据类型：数值型和标称型数据

案例一：自适应算法实现

- 导入数据：

```
def load_sim_data():
    """
    测试数据，
    :return: data_arr   feature对应的数据集
             label_arr  feature对应的分类标签
    """
    data_mat = np.matrix([[1.0, 2.1],
                          [2.0, 1.1],
                          [1.3, 1.0],
                          [1.0, 1.0],
                          [2.0, 1.0]])
    class_labels = [1.0, 1.0, -1.0, -1.0, 1.0]
    return data_mat, class_labels
```

- 算法测试：将符合条件的数据转化,测试是否有某个值小于或者大于我们正在测试的阈值,如果大于某个阈值就是错误的

```
def stump_classify(data_mat, dimen, thresh_val,
thresh_ineq):
    """
    (将数据集，按照feature列的value进行 二分法切分比较来赋值分类)
    :param data_mat: Matrix数据集
    :param dimen: 特征的哪一个列
    :param thresh_val: 特征列要比较的值
    :param thresh_ineq:
    :return: np.array
    """
    ret_array = np.ones((np.shape(data_mat)[0], 1))
    # data_mat[:, dimen] 表示数据集中第dimen列的所有值
    # thresh_ineq == 'lt'表示修改左边的值，gt表示修改右边的值
```

```
    # （这里其实我建议理解为转换左右边，就是一棵树的左右孩子，可能有
点问题。。。待考证）
    if thresh_ineq == 'lt':# 假设左边比较一下
        ret_array[data_mat[:, dimen] <= thresh_val] = -1.0
    else:# 假设右边比较一下
        ret_array[data_mat[:, dimen] > thresh_val] = -1.0
    return ret_array
```

- 单层决策树的实现：最优化单层决策树

```
# 这个算法是为了寻找最好的单层决策树
def build_stump(data_arr, class_labels, D):
    """
    得到决策树的模型 (这个比较重要，需要看懂）
    :param data_arr: 特征标签集合
    :param class_labels: 分类标签集合
    :param D: 最初的特征权重值
    :return: best_Stump     最优的分类器模型
            min_error      错误率
            best_class_est  训练后的结果集
    """
    data_mat = np.mat(data_arr)
    label_mat = np.mat(class_labels).T

    m, n = np.shape(data_mat)
    num_steps = 10.0
    best_stump = {}
    best_class_est = np.mat(np.zeros((m, 1)))#训练后的结果集
    # 无穷大
    min_err = np.inf
    for i in range(n):
        range_min = data_mat[:, i].min()
        range_max = data_mat[:, i].max()
        step_size = (range_max - range_min) / num_steps
        for j in range(-1, int(num_steps) + 1):
            for inequal in ['lt', 'gt']:
                thresh_val = (range_min + float(j) *
step_size)
                predicted_vals = stump_classify(data_mat,
i, thresh_val, inequal)
                err_arr = np.mat(np.ones((m, 1)))
                err_arr[predicted_vals == label_mat] = 0
                # 这里是矩阵乘法
                weighted_err = D.T * err_arr
                '''
                dim             表示 feature列
                thresh_val       表示树的分界值
                inequal         表示计算树左右颠倒的错误率的情况
                weighted_error  表示整体结果的错误率
                best_class_est     预测的最优结果 （与
class_labels对应）
                '''
```

```
                # print('split: dim {}, thresh {}, thresh
inequal: {}, the weighted err is {}'.format(
                #     i, thresh_val, inequal, weighted_err
                # ))
                if weighted_err < min_err:
                    min_err = weighted_err
                    best_class_est =
predicted_vals.copy()#可以保存的结果集储存
                    best_stump['dim'] = i# 第i列
                    best_stump['thresh'] = thresh_val# 阈值
                    best_stump['ineq'] = inequal#比较范围，
是用大于还是用小于
    # best_stump 表示分类器的结果，在第几个列上，用大于 / 小于比较，
阈值是多少 (单个弱分类器)
    # print(best_stump)
    return best_stump, min_err, best_class_est
# print(np.mat(np.ones((5,1))/5))#赋值相同的权重
# [[0.2]
#  [0.2]
#  [0.2]
#  [0.2]
#  [0.2]]
#
print(build_stump(datMat,classLables,np.mat(np.ones((5,1))
/5)))
```

- 构建自适应算法

```
def ada_boost_train_ds(data_arr, class_labels, num_it=40):
    """
    adaBoost训练过程放大
    :param data_arr: 特征标签集合
    :param class_labels: 分类标签集合
    :param num_it: 迭代次数
    :return: weak_class_arr  弱分类器的集合
             agg_class_est   预测的分类结果值
    """
    weak_class_arr = []
    m = np.shape(data_arr)[0]
    # 初始化 D，设置每个特征的权重值，平均分为m份
    D = np.mat(np.ones((m, 1)) / m)
    agg_class_est = np.mat(np.zeros((m, 1)))#初始化0矩阵
    for i in range(num_it):
        # 得到决策树的模型
        best_stump, error, class_est =
build_stump(data_arr, class_labels, D)# 寻找最佳的单层决策树
        print('D: {}'.format(D.T))
        # alpha 目的主要是计算每一个分类器实例的权重(加和就是分类结
果)
        # 计算每个分类器的 alpha 权重值
        alpha = float(0.5 * np.log((1.0 - error) /
max(error, 1e-16)))
```

```
        best_stump['alpha'] = alpha
        # store Stump Params in Array
        weak_class_arr.append(best_stump)
        # print('class_est: {}'.format(class_est.T))
        # 分类正确：乘积为1，不会影响结果，-1主要是下面求e的-alpha
次方
        # 分类错误：乘积为 -1，结果会受影响，所以也乘以 -1
        expon = np.multiply(-1 * alpha *
np.mat(class_labels).T, class_est)
        # 判断正确的，就乘以-1，否则就乘以1， 为什么？ 书上的公式有
问题
        # print('(-1取反)预测值 expon=', expon.T)
        # 计算e的expon次方，然后计算得到一个综合的概率的值
        # 结果发现： 判断错误的样本，D对于的样本权重值会变大。
        # multiply是对应项相乘
        D = np.multiply(D, np.exp(expon))
        D = D / D.sum()
        # 预测的分类结果值，在上一轮结果的基础上，进行加和操作
        # print('叠加前的分类结果class_est:
{}'.format(class_est.T))
        agg_class_est += alpha * class_est
        print('叠加后的分类结果agg_class_est:
{}'.format(agg_class_est.T))
        # sign 判断正为1， 0为0， 负为-1，通过最终加和的权重值，判
断符号。
        # 结果为：错误的样本标签集合，因为是 !=,那么结果就是0 正，1
负，这里1就是表示是错误分辨的
        agg_errors = np.multiply(np.sign(agg_class_est) !=
np.mat(class_labels).T,
                                 np.ones((m, 1)))
        error_rate = agg_errors.sum() / m
        print('total error: {}\n'.format(error_rate))
        if error_rate == 0.0:
            break
    print(D)
    return weak_class_arr
```

- 测试代码

```
def ada_classify(data_to_class, classifier_arr):
    """
    通过刚刚上面那个函数得到的弱分类器的集合进行预测
    :param data_to_class: 数据集
    :param classifier_arr: 分类器列表
    :return: 正负一，也就是表示分类的结果
    """
    data_mat = np.mat(data_to_class)# 测试9

    m = np.shape(data_mat)[0]
    agg_class_est = np.mat(np.zeros((m, 1)))
    for i in range(len(classifier_arr)):
        class_est = stump_classify(
```

```
            data_mat, classifier_arr[i]['dim'],
            classifier_arr[i]['thresh'],
            classifier_arr[i]['ineq']
        )
        agg_class_est += classifier_arr[i]['alpha'] *
class_est
        print(agg_class_est)
    return np.sign(agg_class_est)
```

- 完整代码：

```
# 自适应算法实现实例一
import numpy as np


def load_sim_data():
    """
    测试数据，
    :return: data_arr   feature对应的数据集
             label_arr  feature对应的分类标签
    """
    data_mat = np.matrix([[1.0, 2.1],
                          [2.0, 1.1],
                          [1.3, 1.0],
                          [1.0, 1.0],
                          [2.0, 1.0]])
    class_labels = [1.0, 1.0, -1.0, -1.0, 1.0]
    return data_mat, class_labels
datMat,classLables = load_sim_data()
def stump_classify(data_mat, dimen, thresh_val,
thresh_ineq):
    """
    (将数据集，按照feature列的value进行 二分法切分比较来赋值分类)
    :param data_mat: Matrix数据集
    :param dimen: 特征的哪一个列
    :param thresh_val: 特征列要比较的值
    :param thresh_ineq:
    :return: np.array
    """
    ret_array = np.ones((np.shape(data_mat)[0], 1))
    # data_mat[:, dimen] 表示数据集中第dimen列的所有值
    # thresh_ineq == 'lt'表示修改左边的值，gt表示修改右边的值
    # （这里其实我建议理解为转换左右边，就是一棵树的左右孩子，可能有
点问题。。。待考证）
    if thresh_ineq == 'lt':# 假设左边比较一下
        ret_array[data_mat[:, dimen] <= thresh_val] = -1.0
    else:# 假设右边比较一下
        ret_array[data_mat[:, dimen] > thresh_val] = -1.0
    return ret_array
# 这个算法是为了寻找最好的单层决策树
def build_stump(data_arr, class_labels, D):
    """
```

```
    得到决策树的模型（这个比较重要，需要看懂）
    :param data_arr: 特征标签集合
    :param class_labels: 分类标签集合
    :param D: 最初的特征权重值
    :return: best_Stump     最优的分类器模型
            min_error      错误率
            best_class_est  训练后的结果集
    """
    data_mat = np.mat(data_arr)
    label_mat = np.mat(class_labels).T

    m, n = np.shape(data_mat)
    num_steps = 10.0
    best_stump = {}
    best_class_est = np.mat(np.zeros((m, 1)))#训练后的结果集
    # 无穷大
    min_err = np.inf
    for i in range(n):
        range_min = data_mat[:, i].min()
        range_max = data_mat[:, i].max()
        step_size = (range_max - range_min) / num_steps
        for j in range(-1, int(num_steps) + 1):
            for inequal in ['lt', 'gt']:
                thresh_val = (range_min + float(j) *
step_size)
                predicted_vals = stump_classify(data_mat,
i, thresh_val, inequal)
                err_arr = np.mat(np.ones((m, 1)))
                err_arr[predicted_vals == label_mat] = 0
                # 这里是矩阵乘法
                weighted_err = D.T * err_arr
                '''
                dim             表示 feature列
                thresh_val       表示树的分界值
                inequal         表示计算树左右颠倒的错误率的情况
                weighted_error  表示整体结果的错误率
                best_class_est     预测的最优结果 （与
class_labels对应）
                '''
                # print('split: dim {}, thresh {}, thresh
inequal: {}, the weighted err is {}'.format(
                #     i, thresh_val, inequal, weighted_err
                # ))
                if weighted_err < min_err:
                    min_err = weighted_err
                    best_class_est =
predicted_vals.copy()#可以保存的结果集储存
                    best_stump['dim'] = i# 第i列
                    best_stump['thresh'] = thresh_val# 阈值
                    best_stump['ineq'] = inequal#比较范围，
是用大于还是用小于
```

```
    # best_stump 表示分类器的结果，在第几个列上，用大于 / 小于比较，
阈值是多少 (单个弱分类器)
    # print(best_stump)
    return best_stump, min_err, best_class_est
# print(np.mat(np.ones((5,1))/5))#赋值相同的权重
# [[0.2]
#  [0.2]
#  [0.2]
#  [0.2]
#  [0.2]]
#
print(build_stump(datMat,classLables,np.mat(np.ones((5,1))
/5)))
def ada_boost_train_ds(data_arr, class_labels, num_it=40):
    """
    adaBoost训练过程放大
    :param data_arr: 特征标签集合
    :param class_labels: 分类标签集合
    :param num_it: 迭代次数
    :return: weak_class_arr  弱分类器的集合
             agg_class_est   预测的分类结果值
    """
    weak_class_arr = []
    m = np.shape(data_arr)[0]
    # 初始化 D，设置每个特征的权重值，平均分为m份
    D = np.mat(np.ones((m, 1)) / m)
    agg_class_est = np.mat(np.zeros((m, 1)))#初始化0矩阵
    for i in range(num_it):
        # 得到决策树的模型
        best_stump, error, class_est =
build_stump(data_arr, class_labels, D)# 寻找最佳的单层决策树
        print('D: {}'.format(D.T))
        # alpha 目的主要是计算每一个分类器实例的权重(加和就是分类结
果)
        # 计算每个分类器的 alpha 权重值
        alpha = float(0.5 * np.log((1.0 - error) /
max(error, 1e-16)))
        best_stump['alpha'] = alpha
        # store Stump Params in Array
        weak_class_arr.append(best_stump)
        # print('class_est: {}'.format(class_est.T))
        # 分类正确：乘积为1，不会影响结果，-1主要是下面求e的-alpha
次方
        # 分类错误：乘积为 -1，结果会受影响，所以也乘以 -1
        expon = np.multiply(-1 * alpha *
np.mat(class_labels).T, class_est)
        # 判断正确的，就乘以-1，否则就乘以1，  为什么？  书上的公式有
问题
        # print('(-1取反)预测值 expon=', expon.T)
        # 计算e的expon次方，然后计算得到一个综合的概率的值
        # 结果发现：  判断错误的样本，D对于的样本权重值会变大。
        # multiply是对应项相乘
```

```
        D = np.multiply(D, np.exp(expon))
        D = D / D.sum()
        # 预测的分类结果值，在上一轮结果的基础上，进行加和操作
        # print('叠加前的分类结果class_est:
{}'.format(class_est.T))
        agg_class_est += alpha * class_est
        print('叠加后的分类结果agg_class_est:
{}'.format(agg_class_est.T))
        # sign 判断正为1， 0为0， 负为-1，通过最终加和的权重值，判
断符号。
        # 结果为：错误的样本标签集合，因为是 !=,那么结果就是0 正，1
负，这里1就是表示是错误分辨的
        agg_errors = np.multiply(np.sign(agg_class_est) !=
np.mat(class_labels).T,
                                 np.ones((m, 1)))
        error_rate = agg_errors.sum() / m
        print('total error: {}\n'.format(error_rate))
        if error_rate == 0.0:
            break
    print(D)
    return weak_class_arr
# print(ada_boost_train_ds(datMat,classLables,9))
classLables_arr = ada_boost_train_ds(datMat,classLables,9)

def ada_classify(data_to_class, classifier_arr):
    """
    通过刚刚上面那个函数得到的弱分类器的集合进行预测
    :param data_to_class: 数据集
    :param classifier_arr: 分类器列表
    :return: 正负一，也就是表示分类的结果
    """
    data_mat = np.mat(data_to_class)# 测试9

    m = np.shape(data_mat)[0]
    agg_class_est = np.mat(np.zeros((m, 1)))
    for i in range(len(classifier_arr)):
        class_est = stump_classify(
            data_mat, classifier_arr[i]['dim'],
            classifier_arr[i]['thresh'],
            classifier_arr[i]['ineq']
        )
        agg_class_est += classifier_arr[i]['alpha'] *
class_est
        print(agg_class_est)
    return np.sign(agg_class_est)
# print(ada_classify([0,0],classLables_arr))
```

疝病马数据使用自适应算法实现

- 和上边代码差不多，就是加了一个rank测试

```
import numpy as np
def stump_classify(data_mat, dimen, thresh_val,
thresh_ineq):
    """
    (将数据集，按照feature列的value进行 二分法切分比较来赋值分类)
    :param data_mat: Matrix数据集
    :param dimen: 特征的哪一个列
    :param thresh_val: 特征列要比较的值
    :param thresh_ineq:
    :return: np.array
    """
    ret_array = np.ones((np.shape(data_mat)[0], 1))
    # data_mat[:, dimen] 表示数据集中第dimen列的所有值
    # thresh_ineq == 'lt'表示修改左边的值，gt表示修改右边的值
    # （这里其实我建议理解为转换左右边，就是一棵树的左右孩子，可能有
点问题。。。待考证）
    if thresh_ineq == 'lt':# 假设左边比较一下
        ret_array[data_mat[:, dimen] <= thresh_val] = -1.0
    else:# 假设右边比较一下
        ret_array[data_mat[:, dimen] > thresh_val] = -1.0
    return ret_array
# 这个算法是为了寻找最好的单层决策树
def build_stump(data_arr, class_labels, D):
    """
    得到决策树的模型 (这个比较重要，需要看懂）
    :param data_arr: 特征标签集合
    :param class_labels: 分类标签集合
    :param D: 最初的特征权重值
    :return: best_Stump    最优的分类器模型
            min_error     错误率
            best_class_est  训练后的结果集
    """
    data_mat = np.mat(data_arr)
    label_mat = np.mat(class_labels).T

    m, n = np.shape(data_mat)
    num_steps = 10.0
    best_stump = {}
    best_class_est = np.mat(np.zeros((m, 1)))#训练后的结果集
    # 无穷大
    min_err = np.inf
    for i in range(n):
        range_min = data_mat[:, i].min()
        range_max = data_mat[:, i].max()
        step_size = (range_max - range_min) / num_steps
        for j in range(-1, int(num_steps) + 1):
            for inequal in ['lt', 'gt']:
                thresh_val = (range_min + float(j) *
step_size)
```

```
                predicted_vals = stump_classify(data_mat,
i, thresh_val, inequal)
                err_arr = np.mat(np.ones((m, 1)))
                err_arr[predicted_vals == label_mat] = 0
                # 这里是矩阵乘法
                weighted_err = D.T * err_arr
                '''
                dim            表示 feature列
                thresh_val      表示树的分界值
                inequal        表示计算树左右颠倒的错误率的情况
                weighted_error  表示整体结果的错误率
                best_class_est     预测的最优结果 （与
class_labels对应）
                '''
                # print('split: dim {}, thresh {}, thresh
inequal: {}, the weighted err is {}'.format(
                #     i, thresh_val, inequal, weighted_err
                # ))
                if weighted_err < min_err:
                    min_err = weighted_err
                    best_class_est =
predicted_vals.copy()#可以保存的结果集储存
                    best_stump['dim'] = i# 第i列
                    best_stump['thresh'] = thresh_val# 阈值
                    best_stump['ineq'] = inequal#比较范围，
是用大于还是用小于
    # best_stump 表示分类器的结果，在第几个列上，用大于 / 小于比较，
阈值是多少 (单个弱分类器)
    # print(best_stump)
    return best_stump, min_err, best_class_est
# print(np.mat(np.ones((5,1))/5))#赋值相同的权重
# [[0.2]
#  [0.2]
#  [0.2]
#  [0.2]
#  [0.2]]
#
print(build_stump(datMat,classLables,np.mat(np.ones((5,1))
/5)))
def ada_boost_train_ds(data_arr, class_labels, num_it=40):
    """
    adaBoost训练过程放大
    :param data_arr: 特征标签集合
    :param class_labels: 分类标签集合
    :param num_it: 迭代次数
    :return: weak_class_arr  弱分类器的集合
            agg_class_est    预测的分类结果值
    """
    weak_class_arr = []
    m = np.shape(data_arr)[0]
    # 初始化 D，设置每个特征的权重值，平均分为m份
    D = np.mat(np.ones((m, 1)) / m)
```

```
    agg_class_est = np.mat(np.zeros((m, 1)))#初始化0矩阵
    for i in range(num_it):
        # 得到决策树的模型
        best_stump, error, class_est =
build_stump(data_arr, class_labels, D)# 寻找最佳的单层决策树
        # print('D: {}'.format(D.T))
        # alpha 目的主要是计算每一个分类器实例的权重(加和就是分类结
果)
        # 计算每个分类器的 alpha 权重值
        alpha = float(0.5 * np.log((1.0 - error) /
max(error, 1e-16)))
        best_stump['alpha'] = alpha
        # store Stump Params in Array
        weak_class_arr.append(best_stump)
        # print('class_est: {}'.format(class_est.T))
        # 分类正确：乘积为1，不会影响结果，-1主要是下面求e的-alpha
次方
        # 分类错误：乘积为 -1，结果会受影响，所以也乘以 -1
        expon = np.multiply(-1 * alpha *
np.mat(class_labels).T, class_est)
        # 判断正确的，就乘以-1，否则就乘以1， 为什么？ 书上的公式有
问题
        # print('(-1取反)预测值 expon=', expon.T)
        # 计算e的expon次方，然后计算得到一个综合的概率的值
        # 结果发现： 判断错误的样本，D对于的样本权重值会变大。
        # multiply是对应项相乘
        D = np.multiply(D, np.exp(expon))
        D = D / D.sum()
        # 预测的分类结果值，在上一轮结果的基础上，进行加和操作
        # print('叠加前的分类结果class_est:
{}'.format(class_est.T))
        agg_class_est += alpha * class_est
        # print('叠加后的分类结果agg_class_est:
{}'.format(agg_class_est.T))
        # sign 判断正为1， 0为0， 负为-1，通过最终加和的权重值，判
断符号。
        # 结果为：错误的样本标签集合，因为是 !=,那么结果就是0 正，1
负，这里1就是表示是错误分辨的
        agg_errors = np.multiply(np.sign(agg_class_est) !=
np.mat(class_labels).T,
                                 np.ones((m, 1)))
        error_rate = agg_errors.sum() / m
        print('total error: {}\n'.format(error_rate))
        if error_rate == 0.0:
            break
    # print(D)
    return weak_class_arr
def loadDataSet(fileName):
    numFeat = len(open(fileName).readline().split('\t'))#
22个列 21个特征 1个标签
    dataMat = [];labelMat = []
    fr = open(fileName)
```

```
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        # 将21个特征保存起来
        for i in range(numFeat-1):
            lineArr.append((float(curLine[i])))
        # print(len(lineArr))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat,labelMat
datArr,labelArr = loadDataSet(r'/home/ach/桌面/machine-
learning/ai/Adaboost/7.AdaBoost/horseColicTraining2.txt')
# print(ada_boost_train_ds(datArr,labelArr,10))
classifiterArray =
ada_boost_train_ds(datArr,labelArr,10000)
def ada_classify(data_to_class, classifier_arr):
    """
    通过刚刚上面那个函数得到的弱分类器的集合进行预测
    :param data_to_class: 数据集
    :param classifier_arr: 分类器列表
    :return: 正负一，也就是表示分类的结果
    """
    data_mat = np.mat(data_to_class)# 测试9

    m = np.shape(data_mat)[0]
    agg_class_est = np.mat(np.zeros((m, 1)))
    for i in range(len(classifier_arr)):
        class_est = stump_classify(
            data_mat, classifier_arr[i]['dim'],
            classifier_arr[i]['thresh'],
            classifier_arr[i]['ineq']
        )
        agg_class_est += classifier_arr[i]['alpha'] *
class_est
        # print(agg_class_est)
    return np.sign(agg_class_est)
def test():
    testArr,testLabelArr = loadDataSet(r'/home/ach/桌
面/machine-
learning/ai/Adaboost/7.AdaBoost/horseColicTest2.txt')
    pridict = ada_classify(testArr,classifiterArray)
    errArr = np.mat(np.ones((67,1)))
    error = errArr[pridict!=np.mat(testLabelArr).T].sum()
    print('error = {}'.format(error/len(testLabelArr)))
test()
```

- 和书上结果差不多：

表7-1 不同弱分类器数目情况下的AdaBoost测试和分类错误率。该数据集是个难数据集。通常情况下，AdaBoost会达到一个稳定的测试错误率，而并不会随分类器数目的增多而提高

分类器数目	训练错误率（%）	测试错误率（%）
1	0.28	0.27
10	0.23	0.24
50	0.19	0.21
100	0.19	0.22
500	0.16	0.25
1000	0.14	0.31
10000	0.11	0.33

https://blog.csdn.net/weixin_42479155

处理非均衡问题

- 代码

```
import numpy as np
def stump_classify(data_mat, dimen, thresh_val,
thresh_ineq):
    """
    (将数据集，按照feature列的value进行 二分法切分比较来赋值分类)
    :param data_mat: Matrix数据集
    :param dimen: 特征的哪一个列
    :param thresh_val: 特征列要比较的值
    :param thresh_ineq:
    :return: np.array
    """
    ret_array = np.ones((np.shape(data_mat)[0], 1))
    # data_mat[:, dimen] 表示数据集中第dimen列的所有值
    # thresh_ineq == 'lt'表示修改左边的值，gt表示修改右边的值
    # （这里其实我建议理解为转换左右边，就是一棵树的左右孩子，可能有
点问题。。。待考证）
    if thresh_ineq == 'lt':# 假设左边比较一下
        ret_array[data_mat[:, dimen] <= thresh_val] = -1.0
    else:# 假设右边比较一下
        ret_array[data_mat[:, dimen] > thresh_val] = -1.0
    return ret_array
# 这个算法是为了寻找最好的单层决策树
def build_stump(data_arr, class_labels, D):
    """
    得到决策树的模型 (这个比较重要，需要看懂）
    :param data_arr: 特征标签集合
    :param class_labels: 分类标签集合
    :param D: 最初的特征权重值
    :return: best_Stump     最优的分类器模型
             min_error      错误率
             best_class_est  训练后的结果集
    """
    data_mat = np.mat(data_arr)
    label_mat = np.mat(class_labels).T

    m, n = np.shape(data_mat)
    num_steps = 10.0
    best_stump = {}
    best_class_est = np.mat(np.zeros((m, 1)))#训练后的结果集
    # 无穷大
```

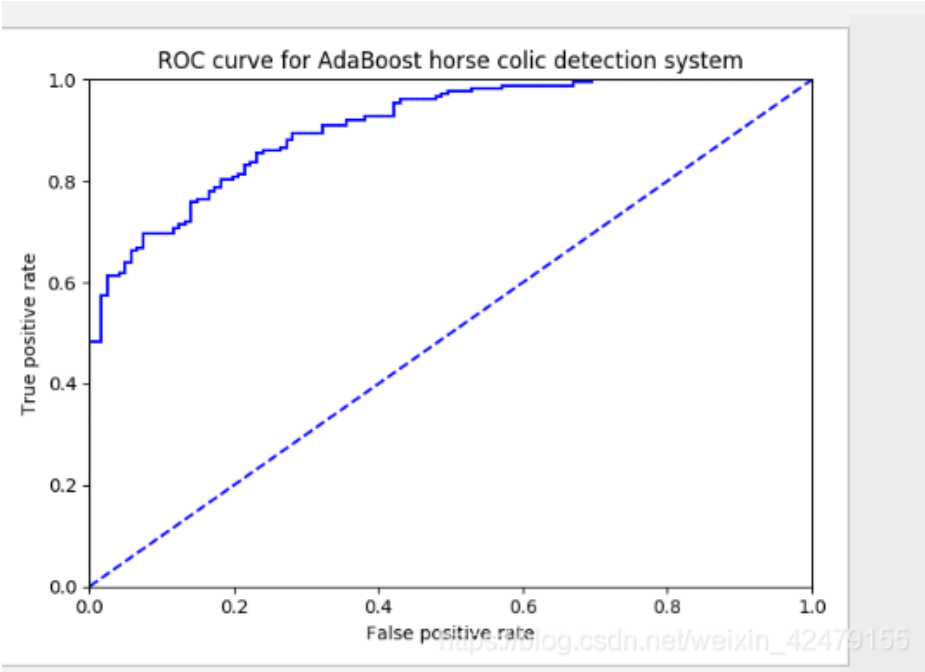
```
    min_err = np.inf
    for i in range(n):
        range_min = data_mat[:, i].min()
        range_max = data_mat[:, i].max()
        step_size = (range_max - range_min) / num_steps
        for j in range(-1, int(num_steps) + 1):
            for inequal in ['lt', 'gt']:
                thresh_val = (range_min + float(j) *
step_size)
                predicted_vals = stump_classify(data_mat,
i, thresh_val, inequal)
                err_arr = np.mat(np.ones((m, 1)))
                err_arr[predicted_vals == label_mat] = 0
                # 这里是矩阵乘法
                weighted_err = D.T * err_arr
                '''
                dim            表示 feature列
                thresh_val      表示树的分界值
                inequal        表示计算树左右颠倒的错误率的情况
                weighted_error  表示整体结果的错误率
                best_class_est    预测的最优结果 （与
class_labels对应）
                '''
                # print('split: dim {}, thresh {}, thresh
inequal: {}, the weighted err is {}'.format(
                #     i, thresh_val, inequal, weighted_err
                # ))
                if weighted_err < min_err:
                    min_err = weighted_err
                    best_class_est =
predicted_vals.copy()#可以保存的结果集储存
                    best_stump['dim'] = i# 第i列
                    best_stump['thresh'] = thresh_val# 阈值
                    best_stump['ineq'] = inequal#比较范围，
是用大于还是用小于
    # best_stump 表示分类器的结果，在第几个列上，用大于 / 小于比较，
阈值是多少 (单个弱分类器)
    # print(best_stump)
    return best_stump, min_err, best_class_est
# print(np.mat(np.ones((5,1))/5))#赋值相同的权重
# [[0.2]
#  [0.2]
#  [0.2]
#  [0.2]
#  [0.2]]
#
print(build_stump(datMat,classLables,np.mat(np.ones((5,1))
/5)))
def ada_boost_train_ds(data_arr, class_labels, num_it=40):
    """
    adaBoost训练过程放大
    :param data_arr: 特征标签集合
```

```
    :param class_labels: 分类标签集合
    :param num_it: 迭代次数
    :return: weak_class_arr  弱分类器的集合
             agg_class_est   预测的分类结果值
    """
    weak_class_arr = []
    m = np.shape(data_arr)[0]
    # 初始化 D，设置每个特征的权重值，平均分为m份
    D = np.mat(np.ones((m, 1)) / m)
    agg_class_est = np.mat(np.zeros((m, 1)))#初始化0矩阵
    for i in range(num_it):
        # 得到决策树的模型
        best_stump, error, class_est =
build_stump(data_arr, class_labels, D)# 寻找最佳的单层决策树
        # print('D: {}'.format(D.T))
        # alpha 目的主要是计算每一个分类器实例的权重(加和就是分类结
果)
        # 计算每个分类器的 alpha 权重值
        alpha = float(0.5 * np.log((1.0 - error) /
max(error, 1e-16)))
        best_stump['alpha'] = alpha
        # store Stump Params in Array
        weak_class_arr.append(best_stump)
        # print('class_est: {}'.format(class_est.T))
        # 分类正确：乘积为1，不会影响结果，-1主要是下面求e的-alpha
次方
        # 分类错误：乘积为 -1，结果会受影响，所以也乘以 -1
        expon = np.multiply(-1 * alpha *
np.mat(class_labels).T, class_est)
        # 判断正确的，就乘以-1，否则就乘以1，  为什么？  书上的公式有
问题
        # print('(-1取反)预测值 expon=', expon.T)
        # 计算e的expon次方，然后计算得到一个综合的概率的值
        # 结果发现：  判断错误的样本，D对于的样本权重值会变大。
        # multiply是对应项相乘
        D = np.multiply(D, np.exp(expon))
        D = D / D.sum()
        # 预测的分类结果值，在上一轮结果的基础上，进行加和操作
        # print('叠加前的分类结果class_est:
{}'.format(class_est.T))
        agg_class_est += alpha * class_est
        # print('叠加后的分类结果agg_class_est:
{}'.format(agg_class_est.T))
        # sign 判断正为1，  0为0，  负为-1，通过最终加和的权重值，判
断符号。
        # 结果为：错误的样本标签集合，因为是 !=,那么结果就是0 正，1
负，这里1就是表示是错误分辨的
        agg_errors = np.multiply(np.sign(agg_class_est) !=
np.mat(class_labels).T,
                                 np.ones((m, 1)))
        error_rate = agg_errors.sum() / m
        print('total error: {}\n'.format(error_rate))
```

```
        if error_rate == 0.0:
            break
    # print(D)
    return weak_class_arr,agg_class_est
def loadDataSet(fileName):
    numFeat = len(open(fileName).readline().split('\t'))#
22个列 21个特征 1个标签
    dataMat = [];labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        # 将21个特征保存起来
        for i in range(numFeat-1):
            lineArr.append((float(curLine[i])))
        # print(len(lineArr))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat,labelMat
datArr,labelArr = loadDataSet(r'/home/ach/桌面/machine-
learning/ai/Adaboost/7.AdaBoost/horseColicTraining2.txt')
# print(ada_boost_train_ds(datArr,labelArr,10))
classifiterArray,aggClassEst =
ada_boost_train_ds(datArr,labelArr,100)
def plot_roc(pred_strengths, class_labels):
    """
    (打印ROC曲线，并计算AUC的面积大小)
    :param pred_strengths: 最终预测结果的权重值
    :param class_labels: 原始数据的分类结果集
    :return:
    """
    import matplotlib.pyplot as plt
    # variable to calculate AUC
    y_sum = 0.0
    # 对正样本的进行求和
    num_pos_class = np.sum(np.array(class_labels) == 1.0)
    # 正样本的概率
    y_step = 1 / float(num_pos_class)
    # 负样本的概率
    x_step = 1 / float(len(class_labels) - num_pos_class)
    # np.argsort函数返回的是数组值从小到大的索引值
    # get sorted index, it's reverse
    sorted_indicies = pred_strengths.argsort()
    # 测试结果是否是从小到大排列
    # 可以选择打印看一下
    # 开始创建模版对象
    fig = plt.figure()
    fig.clf()
    ax = plt.subplot(111)
    # cursor光标值
    cur = (1.0, 1.0)
```

```
    # loop through all the values, drawing a line segment
at each point
    for index in sorted_indicies.tolist()[0]:
        if class_labels[index] == 1.0:
            del_x = 0
            del_y = y_step
        else:
            del_x = x_step
            del_y = 0
            y_sum += cur[1]
        # draw line from cur to (cur[0]-delX, cur[1]-delY)
        # 画点连线 (x1, x2, y1, y2)
        # print cur[0], cur[0]-delX, cur[1], cur[1]-delY
        ax.plot([cur[0], cur[0] - del_x], [cur[1], cur[1]
- del_y], c='b')
        cur = (cur[0] - del_x, cur[1] - del_y)
    # 画对角的虚线线
    ax.plot([0, 1], [0, 1], 'b--')
    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate')
    plt.title('ROC curve for AdaBoost horse colic
detection system')
    # 设置画图的范围区间 (x1, x2, y1, y2)
    ax.axis([0, 1, 0, 1])
    plt.show()
    '''
    参考说明:
http://blog.csdn.net/wenyusuran/article/details/39056013
    为了计算 AUC ，我们需要对多个小矩形的面积进行累加。
    这些小矩形的宽度是x_step，因此可以先对所有矩形的高度进行累加，最
后再乘以x_step得到其总面积。
    所有高度的和(y_sum)随着x轴的每次移动而渐次增加。
    '''
    print("the Area Under the Curve is: ", y_sum * x_step)
plot_roc(aggClassEst.T,labelArr)
```

- 运行结果：



Skleanring实现adaboost算法

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import  DecisionTreeClassifier
import numpy as np
def loadDataSet(fileName):
    numFeat = len(open(fileName).readline().split('\t'))# 22个列 21
个特征 1个标签
    dataMat = [];labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        # 将21个特征保存起来
        for i in range(numFeat-1):
            lineArr.append((float(curLine[i])))
        # print(len(lineArr))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat,labelMat
datArr,labelArr = loadDataSet(r'/home/ach/桌面/machine-
learning/ai/Adaboost/7.AdaBoost/horseColicTraining2.txt')
clf =
AdaBoostClassifier(DecisionTreeClassifier(max_depth=2,min_samples_s
plit=5,min_samples_leaf=5),n_estimators=40,random_state=10000)
clf.fit(np.mat(datArr),labelArr)
datArr1,labelArr1 = loadDataSet(r'/home/ach/桌面/machine-
learning/ai/Adaboost/7.AdaBoost/horseColicTest2.txt')
print(clf.score(np.mat(datArr1),labelArr1))
```