# 《数据挖掘导论》实验 3: 决策树分类实验

# 一、实验目的

（1）了解决策树分类的思想

（2）掌握决策树分类的应用

# 二、实验环境

（1）Anaconda2 开发环境

（2）IDE是ipython notebook

（3）使用的库有numpy，pandas，matplotlib

# 三、实验内容

## 1. 决策树

**（1）决策树简介**

决策树是附加概率结果的一个树状的决策图，是直观的运用统计概率分析的图法。机器学习中决策树是一个预测模型，它表示对象属性和对象值之间的一种映射，树中的每一个节点表示对象属性的判断条件，其分支表示符合节点条件的对象。树的叶子节点表示对象所属的预测结果。

发送邮件域名地址为：

myEmployer.com

否

是

包含单词“曲棍球”的邮件

无聊时需要

阅读邮件

否

是

需要及时处理的朋友邮件

无需阅读的垃圾邮件

如上图所示是一棵简单的决策树，长方形代表判断模块，椭圆形代表终止模块，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，它可以到达另一个判断模块或者终止模块。该图表示的是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，则检查邮件内容里面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”。

**（2）决策树构造**

决策树的构造通常包括3个步骤：特征选择；决策树生成；决策树减枝；如何选择特征有着很多不同量化评估标准标准，从而衍生出ID3、CART和C4.5等算法。本次实验会给出ID3算法3个步骤的理论解释，代码实现上不关注决策树减枝，减枝部分在《机器学习实战》第九章有专门讲解。

1. 特征选择

特征选择是选择有较强分类能力的特征。分类能力通过信息增益或者增益比来刻画。选择特征的标准是找出局部最优的特征作为判断进行切分，取决于切分节点数据集合中类别的有序程度（纯度），划分后的分区数据越纯，切分规则越合适。衡量节点数据纯度的有：熵、基尼系数和方差。熵和基尼系数是针对分类的，方差是针对回归的。

熵是随机变量不确定性的一种度量。设X是离散随机变量，其概率分布为：

则随机变量X的熵为：

其中，定义。

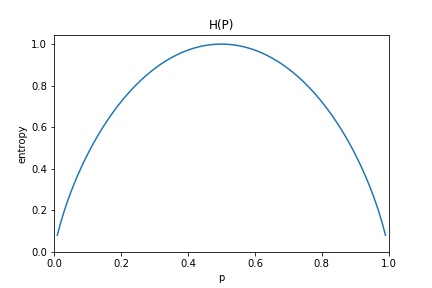
当随机变量只取两个值时，的分布为：

此时熵为：

当或者时候，熵最小（为0），此时随机变量不确定性最小。

当时，熵最大（为1），此时随机变量不确定性最大。

熵entropy H(p)的函数图像如下图所示。



设随机变量(X,Y)，其联合分布为：。则条件熵定义为：

其中，

当熵中的概率由数据估计得到时，称之为经验熵。

当条件熵中的概率由数据估计得到时，称之为经验条件熵。

对于数据集D，我们通过来刻画数据集D的不确定程度。当数据集D中的所有样本都是同一种类别时，。也将记作。给定特征A和训练数据集D，定义信息增益。

信息增益刻画的是由于特征A而使得对数据集D的分类的不确定性减少的程度。构建决策树选择信息增益最大的特征来划分数据集。

假设训练数据集为D，N为其训练数据集容量。假设有K个类别依次为。设为属于类的样本个数。

设特征A是离散的，且有n个不同的取值：,根据特征A的取值将D划分出n个子集：，为对应的中的样本个数。

设集合中属于类的样本集合为：，其容量为：，信息增益算法如下。

* 输入
* 训练数据集D。
* 特征A。
* 输出：信息增益g(D,A)
* 算法步骤
* 根据下式计算数据集D的经验熵H(D)。它就是训练数据集D中，分类Y的概率估计计算得到的经验熵。
* 根据下式计算特征A对于数据集D的经验熵H(D|A)。它使用了特征A的概率估计：，以及经验条件熵：（其中使用了条件概率估计）,意义是：在子集中，Y的分布）
* 根据下式计算信息增益：

熵越大，则表示越混乱；熵越小，表示越有序。因此信息增益表示混乱的减少程度（或者说有序的增加程度）。划分数据集时，每次选择能使得熵减少最多的特征进行数集划分。

1. 决策树生成

ID3生成算法应用信息增益准则选择特征，其本实验的算法描述如下。

* 输入
* 训练数据集D。
* 特征集。
* 输出
* 决策树T。
* 算法步骤
* 若D中所有实例均属于同一类，则T为单节点树，并将作为该节点的类标记，返回T。这是一种特殊情况：D的分类集合只有一个分类。
* 若，则T为单节点树，将D中实例数最大的类作为该节点的类标记，返回T（即多数表决）。这是一种特殊情况：D的特征集合为空。
* 否则计算，其中为特征集合中的各个特征，选择信息增益最大的特征。
* 对特征的每个可能取值，根据将D划分为若干个非空子集，将中实例数最大的类作为标记，构建子节点数，由子节点及其子节点构成树T，返回T。
* 对第i个子节点，以为训练集，以A-{}为特征集，递归调用前面的步骤，取得子树，返回。

**（3）决策树减枝**

决策树需要减枝的原因是：决策树生成算法生成的树对训练数据的预测很准确，但是对于未知的数据分类很差，这就是产生过拟合的现象。其实，原理是一样的，决策树的构建是直到没有特征可选或者信息增益很小，这就导致构建的决策树模型过于复杂，而复杂的模型是在训练数据集上建立的，所以对于测试集往往分类的不准确，这就是过拟合。发生过拟合是由于决策树太复杂，解决过拟合的方法是控制模型的复杂度，对于决策树来说是简化模型，称为减枝。

决策树减枝过程是从已生成的决策树上裁剪掉一些子树或者叶节点。剪枝的目标是通过极小化决策树的整体损失函数或者代价函数来实现的。

决策树剪枝的目的是通过剪枝来提高泛化能力的。剪枝的思路就是在决策树对训练数据的预测误差和数据复杂度之间找到一个平衡。

设树T的叶节点个数为,t为树的叶节点，该叶节点有个样本点，其中属于类的样本点有，个。则有：。

令H(t)为叶节点t上的经验熵，为参数，则决策树T的损失函数定义为：

令：

则：，其中为正则化项，表示预测误差。

* 意味着，即每个节点t内的样本都是纯的（即单一的分类，而不是复杂的）。
* 决策树划分得越细，则T的叶子节点越多，越大；小于等于样本集的数量，当取等号时，树T的每个叶子节点只有一个样本点。
* 参数控制预测误差与模型复杂度之间的关系：
* 较大的会选择较简单的模型
* 较小的会选择较复杂的模型
* 只考虑训练数据与模型的拟合程度，不考虑模型复杂度。

剪枝算法的描述如下。

* 输入
* 生成树T
* 参数
* 输出
* 剪枝树。
* 算法步骤如下。

计算每个节点的经验熵；

递归地从树的叶子节点向上回退：

设一组叶节点退回到父节点之前与之后的整棵树分别为与，对应的损失函数值分别为和。若，则进行剪枝并将父节点变成新的叶节点。

* 递归进行上一步，直到不能继续为止，得到损失函数最小的子树。

**（4）决策树构建代码实现**

calcShannonEnt函数用于计算熵值，dataSet参数表示数据集。该函数首先统计该数据集各个类标的数量，然后计算熵值。

|  |
| --- |
| def calcShannonEnt(dataSet):  numEntries = len(dataSet)  labelCounts = {}  for featVec in dataSet: #the the number of unique elements and their occurance  currentLabel = featVec[-1]  if currentLabel not in labelCounts.keys(): labelCounts[currentLabel] = 0  labelCounts[currentLabel] += 1  shannonEnt = 0.0  for key in labelCounts:  prob = float(labelCounts[key])/numEntries  shannonEnt -= prob \* log(prob,2) #log base 2  return shannonEnt |

splitDataSet函数用于划分数据集，待划分的数据集、划分数据集的特征、特征的值（根据该特征值返回数据子集）。Pyhton语言在函数中传递的是列表的引用，在函数内部对列表对象的修改，将会影响该列表对象的整个生命周期。为了消除这个不良影响，我们需要在函数的开始声明一个新列表对象。因为该函数代码在同一数据集上被调用多次，为了不修改原始数据集，创建一个新的列表对象。

|  |
| --- |
| def splitDataSet(dataSet, axis, value):  retDataSet = []  for featVec in dataSet:  if featVec[axis] == value:  reducedFeatVec = featVec[:axis] #chop out axis used for splitting  reducedFeatVec.extend(featVec[axis+1:])  retDataSet.append(reducedFeatVec)  return retDataSet |

chooseBestFeatureToSplit函数实现根据最大信息增益选取特征，划分数据集。在函数中调用的数据集需要满足一定的要求：第一个要求是，数据必须是一种由列表元素组成的列表，而且所有的列表元素都要具有相同的数据长度；第二个要求是，数据的最后一列或者每个实例的最后一个元素是当前实例的类别标签。数据集一旦满足上述要求，我们就可以在函数的第一行判定当前数据集包含多少特征属性。我们无需限定list中的数据类型，它们既可以是数字也可以是字符串，并不影响实际计算。

第3行代码计算的是整个数据集的原始香农熵，我们保存该度量值，用于与划分完之后的数据集计算的熵值进行比较。第1个for循环遍历数据集中的所有特征。使用列表推导来创建新的列表、将数据集中所有第i个特征值或者所有可能存在的值写入这个新的list中。然后使用Python语言原生的集合（set）数据类型。集合数据类型与列表类型相似，不同之处仅在于集合类型中的每个值互不相同。从列表中创建集合是Python语言得到列表中唯一元素值的最快方法。

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，然后计算数据集的新熵值，并对所有唯一特征值得到的熵求和。信息增益是熵的减少或者是数据无序程度的减少。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。

|  |
| --- |
| def chooseBestFeatureToSplit(dataSet):  numFeatures = len(dataSet[0]) - 1 #the last column is used for the labels  baseEntropy = calcShannonEnt(dataSet)  bestInfoGain = 0.0; bestFeature = -1  for i in range(numFeatures): #iterate over all the features  featList = [example[i] for example in dataSet]#create a list of all the examples of this feature  uniqueVals = set(featList) #get a set of unique values  newEntropy = 0.0  for value in uniqueVals:  subDataSet = splitDataSet(dataSet, i, value)  prob = len(subDataSet)/float(len(dataSet))  newEntropy += prob \* calcShannonEnt(subDataSet)  infoGain = baseEntropy - newEntropy #calculate the info gain; ie reduction in entropy  if (infoGain > bestInfoGain): #compare this to the best gain so far  bestInfoGain = infoGain #if better than current best, set to best  bestFeature = i  return bestFeature |

majorityCnt函数使用分类名称的列表，然后创建键值为classList中唯一值的字典，字典存储了classList中每个类标签出现频率，最后利用operator操作键值排序字典，并返回出现次数最多的分类名称。

|  |
| --- |
| import operator  def majorityCnt(classList):  classCount={}  for vote in classList:  if vote not in classCount.keys(): classCount[vote] = 0  classCount[vote] += 1  sortedClassCount = sorted(classCount.iteritems(), key=operator.itemgetter(1), reverse=True)  return sortedClassCount[0][0] |

createTree函数有两个输入参数：数据集和标签列表。标签列表包含了数据集中所有特征的标签，算法本身并不需要这个变量，但是为了给出数据明确的含义，我们将它作为一个输入参数提供。此外前面提到的数据集的要求这里依然需要满足。上述代码首先创建了名为classList的列表变量，其中包含了数据集的所有类标签。队规函数的第一个停止条件是所有的类标签完全相同，则直接返回该类标签。递归函数的第二个终止条件是使用完了所有的特征，仍然不能将数据集划分成仅包含唯一类别的分组。由于第二个条件无法简单返回唯一的类标签，这里使用前面介绍的majorityCnt函数挑选出次数最多的类别作为返回值。

下一步程序开始创建树，这里使用Python语言的字典类型存储树的信息，当然也可以声明特殊的数据类型存储树，但这里完全没有必要。字典变量myTree存储了树的所有信息，这对于其后绘制树形图非常重要。当前数据集选取的最好特征存储在变量bestFeat中，得到列表包含的所有属性值。

最后代码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，得到的返回值将被插入到字典变量myTree中，因此函数终止执行时，字典中将会嵌套很多代表叶子节点信息的字典数据。在解释这个嵌套数据之前，我们先看下循环的第一行subLabels = label[:]，这行代码复制了类标签，并将其存储在新列表变量subLabels中。之所以这样做，是因为在Python语言中函数参数是列表类型时，参数是按照引用方式传递的。为了保证每次调用函数createTree时不改变原始列表的内容，使用新变量subLabels代替原始列表。

|  |
| --- |
| def createTree(dataSet,labels):  classList = [example[-1] for example in dataSet]  if classList.count(classList[0]) == len(classList):  return classList[0]#stop splitting when all of the classes are equal  if len(dataSet[0]) == 1: #stop splitting when there are no more features in dataSet  return majorityCnt(classList)  bestFeat = chooseBestFeatureToSplit(dataSet)  bestFeatLabel = labels[bestFeat]  myTree = {bestFeatLabel:{}}  del(labels[bestFeat])  featValues = [example[bestFeat] for example in dataSet]  uniqueVals = set(featValues)  # print(uniqueVals)  for value in uniqueVals:  subLabels = labels[:] #copy all of labels, so trees don't mess up existing labels  myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value),subLabels)  return myTree |

Classify函数使用决策树执行分类，第一个参数是决策树，第二个参数是所有特征的名字列表，第三个参数是测试样本。依靠训练数构造了决策树之后，我们可以将它用于实际数据的分类。在执行数据分类时，需要使用决策树以及用于构造决策树的标签向量。然后，程序比较测试数据与决策树上的数值，递归执行该过程直到进入叶子节点；最后将测试数据定义为叶子节点所属的类型。

|  |
| --- |
| def classify(inputTree,featLabels,testVec):  firstStr = inputTree.keys()[0]  secondDict = inputTree[firstStr]  featIndex = featLabels.index(firstStr)  key = testVec[featIndex]  valueOfFeat = secondDict[key]  if isinstance(valueOfFeat, dict):  classLabel = classify(valueOfFeat, featLabels, testVec)  else: classLabel = valueOfFeat  return classLabel |

构建决策树是很耗时的任务，及时处理很小的数据集，如前面的样本数据，也要花费几秒钟的时间，如果数据集很大，会耗费很多计算时间。然后用创建好的决策树解决分类问题，则可以很快完成。因此为了节省计算时间，最好能够在每次执行分类时调用已经构造好的决策树。为了解决这个问题，需要使用python模块的pickle序列化对象。序列化对象可以在磁盘上保存对象，并在需要的时候读取出来。任何对象都可以执行序列化操作，字典对象也不例外。

storeTree函数用于保存决策树，grabTree函数用于读取决策树。

|  |
| --- |
| def storeTree(inputTree,filename):  import pickle  fw = open(filename,'w')  pickle.dump(inputTree,fw)  fw.close()    def grabTree(filename):  import pickle  fr = open(filename)  return pickle.load(fr) |

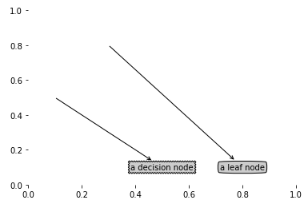
**（5）决策树绘制代码实现**

Matplotlib提供了一个非常有用的注解工具annotations，它可以在数据图形上添加文本注解。

使用文本注解绘制树的节点的例子代码如下表所示。第1行是导入matplotlib的绘图模块。第二行和第三行是设置文本框的格式 。第3行是设置箭头的格式。plotNode是绘制注节点的函数，该函数传入4个参数，nodeTxt是文本框的内容，centerPt是文本框中心坐标，parentPt是箭头的尾部坐标，该函数调用annotate来绘制注解函数。createPlot函数是一个绘图函数，第11行是新建一个figure对象，第12行是清空fig中的内容，第13行是创建一个axes对象，第14和15行调用plotNode绘制不同的节点，第16行显示绘图结果。

|  |
| --- |
| 1.import matplotlib.pyplot as plt  2.decisionNode = dict(boxstyle="sawtooth", fc="0.8")  3.leafNode = dict(boxstyle="round4", fc="0.8")  4.arrow\_args = dict(arrowstyle="<-")  5.def plotNode(nodeTxt, centerPt, parentPt, nodeType):  6. createPlot.ax1.annotate(nodeTxt, xy=parentPt,  7. xycoords='axes fraction',  8. xytext=centerPt, textcoords='axes fraction',  9. va="center", ha="center", bbox=nodeType, arrowprops=arrow\_args)  10.def createPlot():  11. fig = plt.figure(1, facecolor='white')  12. fig.clf()  13. createPlot.ax1 = plt.subplot(111, frameon=False)  14. plotNode('a decision node', (0.15, 0), (0.1, 1), decisionNode)  15. plotNode('a leaf node', (0.8, 0), (0.3, 0.8), leafNode)  16. plt.show() |

运行结果如下图所示，从图中的可以知道，如果想要绘制一棵决策树，就需要计算出节点的坐标。因为该坐标系统是0到1的范围的，那么子树各节点的坐标跟子树的深度和叶子的数量是有关系的，可以将空间等分，然后根据树的规模计算出根节点的坐标，递归的绘制树的各个节点。



在箭头的中点绘制文本信息

|  |
| --- |
| def plotMidText(cntrPt, parentPt, txtString):  xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]  yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]  createPlot.ax1.text(xMid, yMid, txtString) |

计算树的叶子数和深度的代码如下表所示，认真看会发现这两个函数的逻辑是相似的。因为一棵树可以用一个嵌套字典表示，所以如果是一个字典，那么可以认为是一棵子树，否则认为已经到叶子节点了。计算叶子节点的方法是递归到叶子节点，每个叶子节点向父节点返回1，父节点对其子树返回的值进行求和，将和传递给该节点的父节点，逐层向上累加，最后根节点的值便是整棵树的叶子数了。求树的深度也是类似的思路，递归到叶子节点，向上一层返回1（叶子的深度为1），父节点将最深的子树的深度加1作为自己的最大深度，并将自身的最大深度返回给上一层，逐层向上，最终根节点记录的是整棵树的最大深度。

|  |
| --- |
| def getNumLeafs(myTree):  numLeafs = 0  firstStr = myTree.keys()[0]  secondDict = myTree[firstStr]  for key in secondDict.keys():  if type(secondDict[key]).\_\_name\_\_=='dict'  numLeafs += getNumLeafs(secondDict[key])  else: numLeafs +=1  return numLeafs  def getTreeDepth(myTree):  maxDepth = 0  firstStr = 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 |

plotTree递归绘制决策树。树的宽度用于计算放置判断节点的位置，主要的计算原则是将它放在所有叶子节点的中间，而不是仅仅是它的子节点中间，同时我们使用两个全局变量plotTree.xOff和plotTree.yOff追踪已经绘制的节点位置，以及放置下一个节点的恰当位置。另一个需要说明的是，绘制图形的x轴和y轴有效范围是均是0.0到1.0。通过计算树包含的所有叶子数，划分图形的宽度，从而计算得到当前节点的中心位置，也就是说，我们按照叶子节点的数目将x轴划分为若干部分，按照图形比例绘制树形图的最大好处是无需关心实际输出的图形的大小，一旦图形大小发生变化，函数会自动按照图形大小重新绘制。如果以像素为单位绘制图形，则缩放图形就不是一件简单的工作了。

第2行和第3行分别是计算子树的叶子节点数和子树的深度。第4行是获得当前子树的根节点，第5到7行是计算根节点的中心位置，绘制特征值和绘制当前子树的根节点。第8行获取当前子树的所有下一层子树。第9行计算剩余的高度。第10行到第16递归绘制子树，如果不是叶子，递归调用plotTree绘制子树，如果是叶子，则计算坐标并绘制叶子节点。第17行计算父节点的plotTree.yOff，因为这是个全局变量，函数返回时得修改回来，父亲节点需要使用该值。

|  |
| --- |
| 1.def plotTree(myTree, parentPt, nodeTxt):  2. numLeafs = getNumLeafs(myTree) #this determines the x width of this tree  3. depth = getTreeDepth(myTree)  4. firstStr = myTree.keys()[0] #the text label for this node should be this  5. cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)  6. plotMidText(cntrPt, parentPt, nodeTxt)  7. plotNode(firstStr, cntrPt, parentPt, decisionNode)  8. secondDict = myTree[firstStr]  9. plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD  10. for key in secondDict.keys():  11. if type(secondDict[key]).\_\_name\_\_=='dict':#test to see if the nodes are dictonaires, if not they are leaf nodes  12. plotTree(secondDict[key],cntrPt,str(key)) #recursion  13. else: #it's a leaf node print the leaf node  14. plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW  15. plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff), cntrPt, leafNode)  16. plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))  17. plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD |

修改createPlot函数将上面的函数都调用起来，绘制一棵完整的决策树。createPlot会调用plotTree函数，函数plotTree又一次调用了前面介绍的函数。绘制树形图的很多工作都是在函数plotTree中完成的，函数plotTree首先计算树的宽度和高度。全局变量plotTree.totalW存储树的宽度，全局变量plotTree.totalD存储树的深度，我们使用这两个变量计算树节点的摆放位置，这样可以将树绘制在水平方向和垂直方向的中心位置。

|  |
| --- |
| def createPlot(inTree, figsize=None):  fig = plt.figure(1, figsize, facecolor='white')  fig.clf()  axprops = dict(xticks=[], yticks=[])  createPlot.ax1 = plt.subplot(111, frameon=False, \*\*axprops) #no ticks  #createPlot.ax1 = plt.subplot(111, frameon=False) #ticks for demo puropses  plotTree.totalW = float(getNumLeafs(inTree))  plotTree.totalD = float(getTreeDepth(inTree))  plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;  plotTree(inTree, (0.5,1.0), '')  plt.show() |

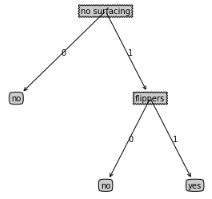
使用下表的函数来创建一棵树用于测试上述代码：

|  |
| --- |
| def retrieveTree(i):  listOfTrees =[{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}},  {'no surfacing': {0: 'no', 1: {'flippers': {0: {'head': {0: 'no', 1: 'yes'}}, 1: 'no'}}}}  ]  return listOfTrees[i] |

使用retrieveTree函数生成一树，然后用createPlot绘制一棵树。

|  |
| --- |
| myTree = retrieveTree(0)  createPlot(myTree) |

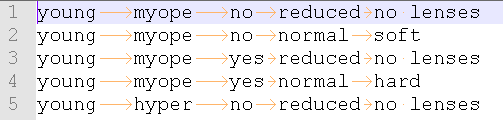
结果如下图所示：



## 2.案例一，使用决策树预测隐形眼镜类型

### (1)加载数据

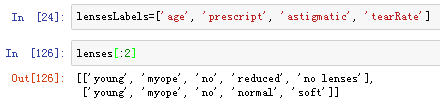
隐形眼镜数据，数据的格式如下图所示，有5列数据，数据之间以’\t’分隔。前四列数据表示特征，字段名分别是’age’，’prescript’ ，’astigmatic’，’tearRate’。最后一列表示隐形眼镜类型。



加载数据

|  |
| --- |
| fr=open('lenses.txt')  lenses=[inst.strip().split('\t') for inst in fr.readlines()]  lensesLabels=['age', 'prescript', 'astigmatic', 'tearRate'] |

输出数据的前两行，结果如下图所示：

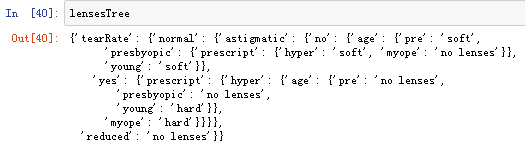


### (2)创建一棵决策树

使用createTree函数创建一棵决策树，因为第二个参数是个用，内部修改时外部相应对象的值也会改变，所以这里使用拷贝。第一行代码是导入决策树模块，第二行代码是导入深拷贝模块。

|  |
| --- |
| import trees  from copy import deepcopy  lensesTree = trees.createTree(lenses,deepcopy(lensesLabels)) |

打印决策树信息如下图所示：

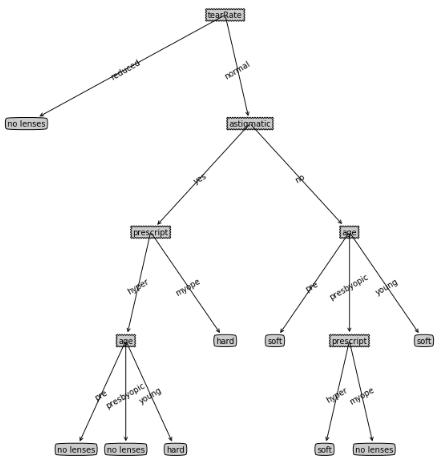


### (3)决策树可视化

createPlot函数用于绘制决策树，这个函数在《机器学习实战》的源码基础上添加了figsize参数用于设置绘图的尺寸。

|  |
| --- |
| import treePlotter  treePlotter.createPlot(lensesTree, figsize=(10,10)) |

决策树绘制的结果如下图所示：



### (4)决策树保存，加载

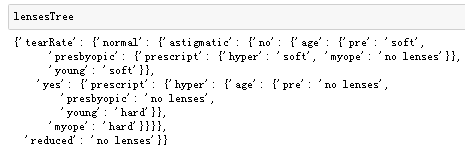
storeTree函数用于保存决策树，第一次参数是决策树，第二个参数是文件名。

|  |
| --- |
| trees.storeTree(lensesTree, 'lensesTree') |

grabTree 函数用于读取决策树，参数是文件名。

|  |
| --- |
| lensesTree = trees.grabTree('lensesTree') |

读取决策后，输出决策树信息如下图所示：



### (5)使用决策树进行预测

得到决策树后，可以使用classify函数利用决策树进行预测，该例子是根据样本信息['young', 'myope', 'no', 'normal', 'soft']，预测隐形眼镜的类型。

|  |
| --- |
| # 使用决策树进行分类  # 样本  asample = ['young', 'myope', 'no', 'normal', 'soft']  # 特征字段名列表  lensesLabels=['age', 'prescript', 'astigmatic', 'tearRate']  trees.classify(lensesTree, lensesLabels, asample) |

结果如下图所示：



## 6.实践任务

**（1）使用决策树预测毒蘑菇**

**a)数据来源:** [**https://archive.ics.uci.edu/ml/datasets/Mushroom**](https://archive.ics.uci.edu/ml/datasets/Mushroom)

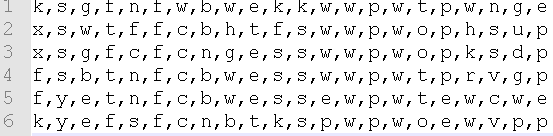
mushrom\_for\_decisionTree文件夹中是整理后的双孢环柄菇数据，共有22个字段，各个字段的含义如下表所示。

|  |  |  |
| --- | --- | --- |
| 1 | cap-shape | bell=b,conical=c,convex=x,flat=f,knobbed=k,sunken=s |
| 2 | cap-surface | fibrous=f,grooves=g,scaly=y,smooth=s |
| 3 | cap-color | brown=n,buff=b,cinnamon=c,gray=g,green=r,pink=p,purple=u,red=e,white=w,yellow=y |
| 4 | bruises? | bruises=t,no=f |
| 5 | odor | almond=a,anise=l,creosote=c,fishy=y,foul=f,musty=m,none=n,pungent=p,spicy=s |
| 6 | gill-attachment | ttached=a,descending=d,free=f,notched=n |
| 7 | gill-spacing | lose=c,crowded=w,distant=d |
| 8 | gill-size | broad=b,narrow=n |
| 9 | gill-color | black=k,brown=n,buff=b,chocolate=h,gray=g,green=r,orange=o,pink=p,purple=u,red=e,white=w,yellow=y |
| 10 | stalk-shape | enlarging=e,tapering=t |
| 11 | stalk-surface-above-ring | fibrous=f,scaly=y,silky=k,smooth=s |
| 12 | stalk-surface-below-ring | fibrous=f,scaly=y,silky=k,smooth=s |
| 13 | stalk-color-above-ring | brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y |
| 14 | stalk-color-below-ring | brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y |
| 15 | veil-type | partial=p,universal=u |
| 16 | veil-color | brown=n,orange=o,white=w,yellow=y |
| 17 | ring-number | none=n,one=o,two=t |
| 18 | ring-type | cobwebby=c,evanescent=e,flaring=f,large=l,none=n,pendant=p,sheathing=s,zone=z |
| 19 | spore-print-color | black=k,brown=n,buff=b,chocolate=h,green=r,orange=o,purple=u,white=w,yellow=y |
| 20 | population | abundant=a,clustered=c,numerous=n,scattered=s,several=v,solitary=y |
| 21 | habitat | grasses=g,leaves=l,meadows=m,paths=p,urban=u,waste=w,woods=d |
| 22 | classes | edible=e, poisonous=p |

数据的格式如下图所示，每个数据以逗号隔开。

文件mushroom\_for\_decisionTree\_train.csv存放的是训练数据。

文件mushroom\_for\_decisionTree\_test.csv存放的是测试数据。



**b)分类评价指标**

**评价指标：**准确度

准确度越高，说明分类的效果越好。

其中a表示预测正确的样本个数，m是总的样本个数。

**c)实验要求:**

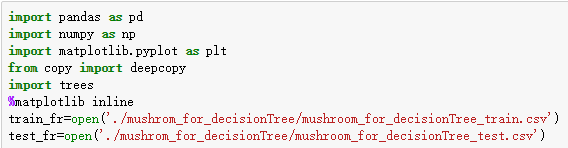
1)使用mushroom\_for\_decisionTree\_train.csv中的数据构建决策树

2)绘制1)中构建的决策树，并保存决策树。

3)加载决策树，并使用mushroom\_for\_decisionTree\_test.csv中的数据测试决策树的分类效果，评价指标使用b)中的准确度。

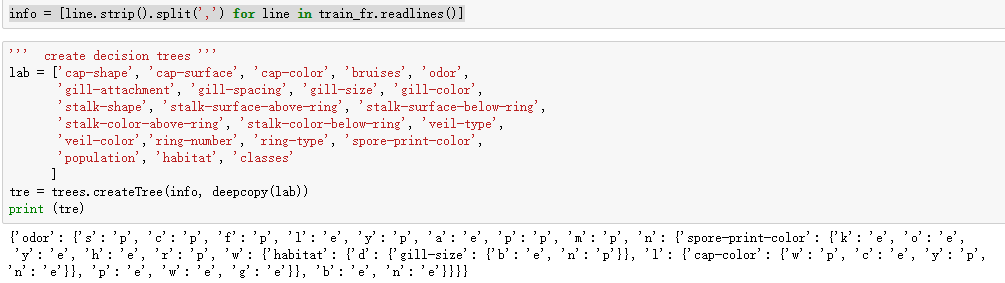
# 四、实验过程

1．引入相应包以及文件



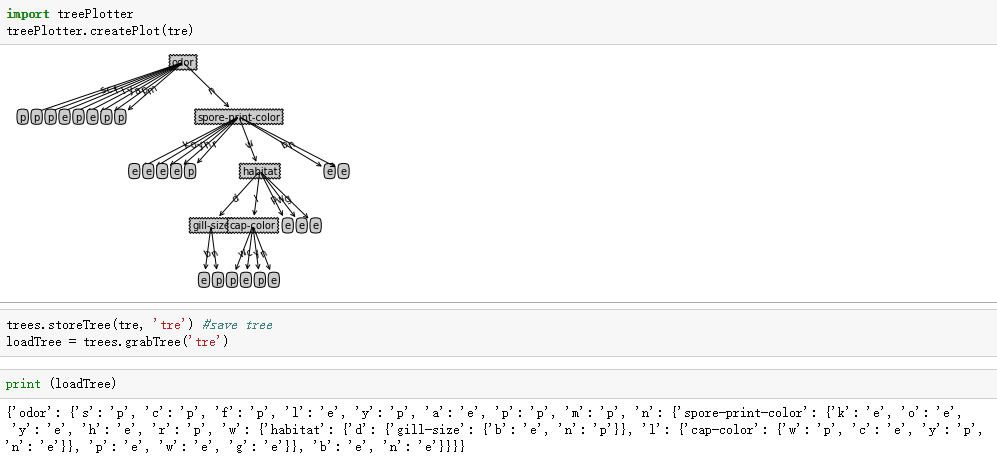
上面一段代码中，引入了pandas、numpy以及matplotlob等绘图库，以及引用了上面编写的trees.py，并打开我们的训练集以及测试集数据。

2．处理数据并完成树的构建



上图将我们的训练集信息抽象成二维数组，并使用trees.py中完成的构建决策树方法完成对决策树的构建。打印结果在上图下方。

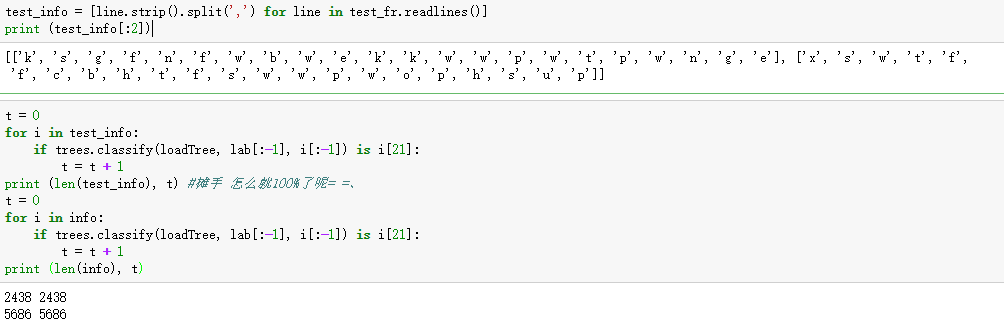
3．打印并保存、加载树



上图我们使用treePlotter.py实现的方法进行我们树的打印，可以见到打印出来的树在两断代码中间。

之后我们使用trees.py中实现的store以及load的方法分别对树进行保存和加载，可以见到加载出来的树与之前生成的树一致。（使用dict存储树可以更方便于格式化成树）。

4．对我们生成树质量的评测



我们先进行测试集的格式化，将其格式化成为二维数组，再进行我们的决策判断。可以看到，训练集的拟合程度为100%，初步怀疑有过拟合现象，但在测试集的测试中，得出的结果也是100%，故而应该没出现过拟合情况。

# 五、实验总结

本次实验初步总结了决策树的思想，基本了解了决策树的应用，并在毒蘑菇预测中进行了预测操作。

期间遇到了一些问题，诸如本机使用的python版本为3，而trees.py以及treeplotter.py中实现的一些方法是为python2中的版本，有所不兼容问题。此处通过修改部分代码完成了此次实验。