1.主成分分析算法实现

1.概念

(1).PCA 的思想是将 n 维特征映射到 k 维上（k<n，降维），这 k 维是全新的正交特征。这 k 维特征称为主元，是重新构造出来的 k 维特征，不是简单地从 n 维特征中去除其余 n-k 维特征。

(2) 降维的必要性

a.多重共线性--预测变量之间相互关联。多重共线性会导致解空间的不稳定，从而可能导致结果的不连贯。

b.过多的变量会妨碍查找规律的建立。

c.仅在变量层面上分析可能会忽略变量之间的潜在联系。例如几个预测变量可能落入仅反映数据某一方面特征的一个组内。

(3) 降维的目的

a.减少预测变量的个数

b.确保这些变量是相互独立的

c.提供一个框架来解释结果

PCA（Principal Component Analysis）不仅仅是对高维数据进行降维，更重要的是经过降维去除了噪声，发现了数据中的模式。

PCA 把原先的 n 个特征用数目更少的 m 个特征取代，新特征是旧特征的线性组合，这些线性组合最大化样本方差，尽量使新的 m 个特征互不相关。从旧特征到新特征的映射捕获数据中的固有变异性

2.算法原理

(1) 数学原理

方差

在二维平面中选择一个方向，将所有数据都投影到这个方向所在直线上，用投影值表示原始记录。这是一个二维降到一维的问题。如何选择这个方向（或者说

基）才能尽量保留最多的原始信息呢？一种直观的看法是：希望投影后的投影值尽可能分散。要让投影后投影值尽可能分散，而这种分散程度，可以用数学上的方差来表述

$$Var(a) = \frac{1}{m}\sum_{i=1}^{m}(a_i - \mu)^2$$

在进行 PCA 之前。必须对每一个特征都进行均值化，则方差可以直接用每个元素的平方和除以元素个数表示：

$$Var(a) = \frac{1}{m}\sum_{i=1}^{m}a_i^2$$

协方差矩阵

对于上面二维降成一维的问题来说，找到那个使得方差最大的方向就可以了。不过对于更高维，还有一个问题需要解决。考虑三维降到二维问题。与之前相同，首先我们希望找到一个方向使得投影后方差最大，这样就完成了第一个方向的选择，继而我们选择第二个投影方向。如果我们还是单纯只选择方差最大的方向，很明显，这个方向与第一个方向应该是"几乎重合在一起"，显然这样的维度是没有用的，因此，应该有其他约束条件。从直观上说，让两个字段尽可能表示更多的原始信息，我们是不希望它们之间存在（线性）相关性的，因为相关性意味着两个字段不是完全独立，必然存在重复表示的信息。

数学上可以用两个字段的协方差表示其相关性，由于已经让每个字段均值为 0，则：

$$Cov(a,b) = \frac{1}{m}\sum_{i=1}^{m}a_i b_i$$

在字段均值为 0 的情况下，两个字段的协方差简洁的表示为其内积除以元素数 m。

至此，我们得到了降维问题的优化目标：将一组 N 维向量降为 K 维（K 大于 0，小于 N），其目标是选择 K 个单位（模为 1）正交基，使得原始数据变换到这组基上后，各字段两两间协方差为 0，而字段的方差则尽可能大（在正交的约束下，取最大的 K 个方差）。

假设我们只有a和b两个字段，那么我们将它们按行组成矩阵X：

$$X = \begin{pmatrix} a_1 & a_2 & \cdots & a_m \\ b_1 & b_2 & \cdots & b_m \end{pmatrix}$$

然后我们用X乘以X的转置，并乘上系数1/m：

$$\frac{1}{m}XX^{\mathsf{T}} = \begin{pmatrix} \frac{1}{m}\sum_{i=1}^{m} a_i^2 & \frac{1}{m}\sum_{i=1}^{m} a_i b_i \\ \frac{1}{m}\sum_{i=1}^{m} a_i b_i & \frac{1}{m}\sum_{i=1}^{m} b_i^2 \end{pmatrix}$$

设原始数据矩阵 X 对应的协方差矩阵为 C，而 P 是一组基按行组成的矩阵，设 Y=PX，则 Y 为 X 对 P 做基变换后的数据。设 Y 的协方差矩阵为 D，我们推导一下 D 与 C 的关系：

$$\begin{aligned} D &= \frac{1}{m}YY^{\mathsf{T}} \\ &= \frac{1}{m}(PX)(PX)^{\mathsf{T}} \\ &= \frac{1}{m}PXX^{\mathsf{T}}P^{\mathsf{T}} \\ &= P(\frac{1}{m}XX^{\mathsf{T}})P^{\mathsf{T}} \\ &= PCP^{\mathsf{T}} \end{aligned}$$

我们要找的 P 不是别的，而是能让原始协方差矩阵对角化的 P。换句话说，优化目标变成了寻找一个矩阵 P，满足 PCP^T是一个对角矩阵，并且对角元素按从大到小依次排列，那么 P 的前 K 行就是要寻找的基，用 P 的前 K 行组成的矩阵乘以 X 就使得 X 从 N 维降到了 K 维并满足上述优化条件。

3.算法源码

首先就是数据集的加载 （数据集为一个 25*2 的矩阵）

```
def getDataSource():
    sets = pd.read_csv("D://pcadataset2.csv", sep=" ")
    dataSet = []
    # print(sets)
    for i in range(len(sets)):
        initDataSet = [x for x in sets.loc[i].tolist()]
        dataSet.append(initDataSet)

    dataMat = np.mat(dataSet,dtype=float)
    dataMatTrans = dataMat.T
    print(dataMatTrans)
    return dataMatTrans
```

矩阵转置之后的每一行进行零均值化

```
#将x的每一行（代表一个属性字段）进行零均值化，也就是减去这一行的均值
def normalize(dataSet):
    # print(dataSet.shape)
    rowLen = dataSet.shape[0]
    columnLen = dataSet.shape[1]
    posOut = 0
    for row in range(rowLen):

        meanValue = np.mean(dataSet[row])

        for column in range(columnLen):
            dataSet[row, column] = dataSet[row, column]-meanValue

    return dataSet,columnLen
```

计算协方差矩阵

```
#计算求得协方差矩阵,并将计算结果返回
def getCovMatrix(dataSet,columnLen):
    dataTransMat = dataSet.T
    multiMatrix = dataSet*dataTransMat
    # print(dataSet)
    # print(dataTransMat)
    for i in range(multiMatrix.shape[0]):
        for j in range(multiMatrix.shape[1]):
            multiMatrix[i, j] = multiMatrix[i,j] / columnLen
    return multiMatrix
```

执行雅克比算法计算特征值和特征向量(返回矩阵对应的特征值和特征向量)

```python
#用雅克比算法计算特征值和特征向量
def getFeatureValueWithJacobi(dataSet):
    #算法迭代的临界条件
    thresHold = 0.001
    #矩阵的行数和列数
    rowLen,columnLen = dataSet.shape
    rows,columns = 0,0
    featVectors, mediaMat = initData(columnLen, rowLen)
    # print(mediaMat)
    while True:

        columns, maxValue, rows = getMatrixValue(columnLen, columns, dataSet, rowLen, rows)

        if maxValue < thresHold:
            break

        cosAlpha,sinAlpha = getAngle(columns, dataSet, rows)

        featVectors = reNewMatrix(columnLen, columns, cosAlpha, dataSet, featVectors, mediaMat, rowLen, rows, sinAlpha)
    return dataSet,featVectors
```

雅克比算法的步骤：

1. 计算矩阵非对角线元素的绝对值的最大值(返回最大值以及它的行号和列号)

```python
#在矩阵的非对角线上找到绝对值最大的元素
def getMatrixValue(columnLen, columns, dataSet, rowLen, rows):

    maxValue = 0
    for i in range(rowLen):
        for j in range(columnLen):
            if i != j:
                absValue = abs(dataSet[i, j])
                if absValue > maxValue:
                    maxValue = absValue
                    # 找到最大元素Apq  p = rows ,q = columns
                    rows = i
                    columns = j
    return columns, maxValue, rows
```

2. 计算平面旋矩阵所要旋转的角度

```python
#计算平面旋矩阵所要旋转的角度
def getAngle(columns, dataSet, rows):
    matPPValue = dataSet[rows, rows]
    matPQValue = dataSet[rows, columns]
    matQQValue = dataSet[columns, columns]
    # print(matPPValue,matPQValue,matQQValue)
    alphaValue = ((-2) * matPQValue) / (matQQValue - matPPValue)
    # 求旋转角度 反 正切函数
    # alpha = (1/2)*m.atan(alphaValue)
    alpha = m.atan(alphaValue) / 2
    sinAlpha = m.sin(alpha)
    cosAlpha = m.cos(alpha)
    sin2Alpha = m.sin(2 * alpha)
    cos2Alpha = m.cos(2 * alpha)
    dataSet[rows, rows] = matPPValue * cosAlpha * cosAlpha + \
                          matQQValue * sinAlpha * sinAlpha + 2 * matPQValue * cosAlpha * sinAlpha
    dataSet[columns, columns] = matPPValue * sinAlpha * sinAlpha + \
                                matQQValue * cosAlpha * cosAlpha - 2 * matPQValue * cosAlpha * sinAlpha
    dataSet[rows, columns] = (1/2) * (matQQValue - matPPValue) * sin2Alpha + matPQValue * cos2Alpha
    dataSet[columns, rows] = dataSet[rows, columns]
    return cosAlpha, sinAlpha
```

3， 第一轮迭代完成之后，对原始矩阵进行更新，

```python
#第一轮迭代完成之后，对原始矩阵进行更新，
def reNewMatrix(columnLen, columns, cosAlpha, dataSet, featVectors, mediaMat, rowLen, rows, sinAlpha):
    for i in range(columnLen):

        if i != rows and i != columns:
            dataSet[rows, i] = sinAlpha * dataSet[columns, i] + cosAlpha * dataSet[rows, i]
            dataSet[columns, i] = cosAlpha * dataSet[columns, i] - sinAlpha * dataSet[rows, i]
    for j in range(rowLen):

        if j != rows and j != columns:
            dataSet[j, rows] = sinAlpha * dataSet[j, columns] + cosAlpha * dataSet[j, rows]
            dataSet[j, columns] = cosAlpha * dataSet[j, columns] - sinAlpha * dataSet[j, rows]

    # 计算特征向量
    mediaMat[rows, rows] = cosAlpha
    mediaMat[rows, columns] = -sinAlpha
    mediaMat[columns, rows] = sinAlpha
    mediaMat[columns, columns] = cosAlpha
    featVectors = featVectors * mediaMat
    return featVectors
```

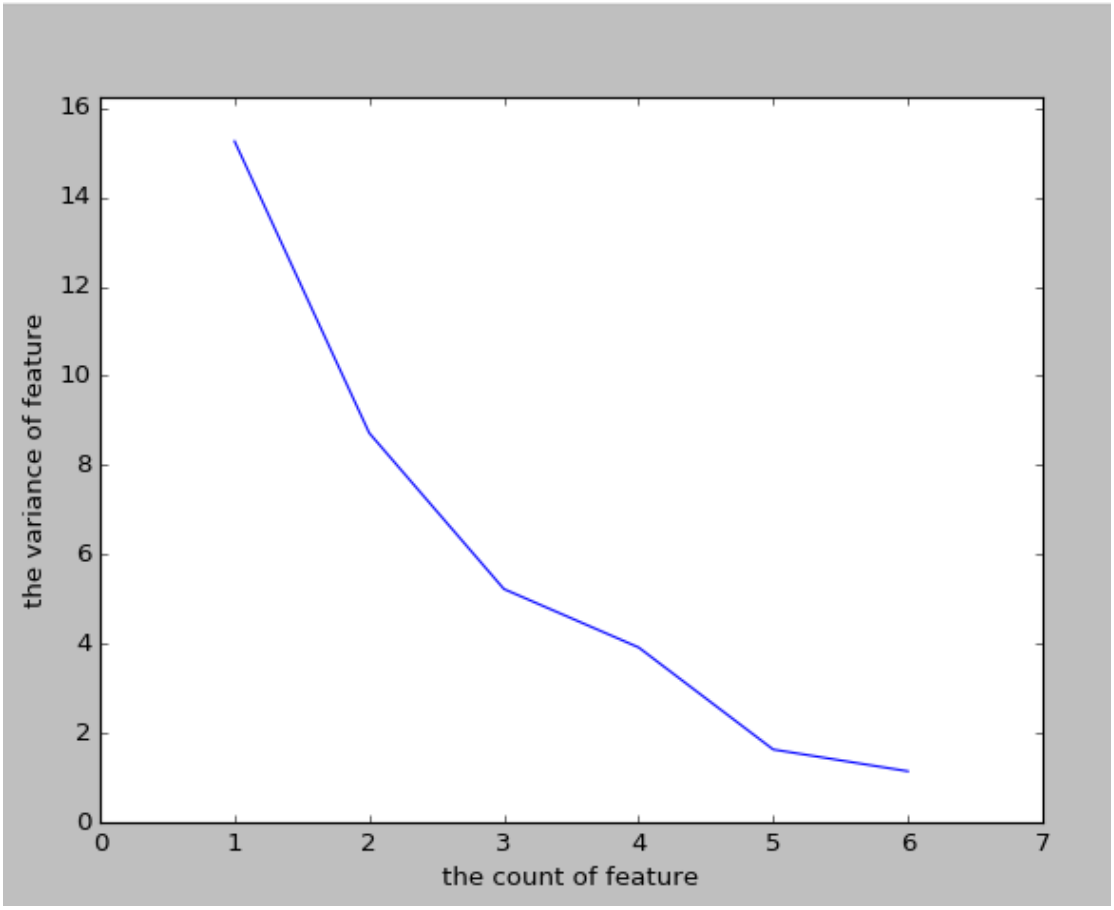
4， 特征向量矩阵的初始化

```python
#特征向量矩阵的初始化
def initData(columnLen, rowLen):
    # 存放特征向量的矩阵
    featVectors = np.zeros((rowLen, columnLen),dtype=float)
    mediaMat = np.zeros((rowLen, columnLen),dtype=float)
    for i in range(rowLen):
        for j in range(columnLen):
            featVectors[i, j] = 1

    # 定义一个中间矩阵mediaMat,每次迭代的时候都需要更新
    for i in range(rowLen):
        for j in range(columnLen):
            mediaMat[i, j] = 1

    return featVectors, mediaMat
```

选择降维后最佳特征的个数 k(利用累计方差百分比计算)如下代码所示，阈值的大小由自己确定(方差所占百分比 70%以上最好)

```
"""
 利用累计方差百分比计算
 若累计方差百分比大于某一个阈值，则可用前面几个特征作为降维之后的特征
"""
def getBestKValueWithPercent(dataSetDiag):
    k = 0
    featValue = dict()
    xAndyValue = []
    slopeList = []
    for i in range(dataSetDiag.shape[0]):
        featValue[dataSetDiag[i,i]] = i
    sortedFeatValue = sorted(featValue,reverse=True)

    lens = len(sortedFeatValue)
    maxValue = sortedFeatValue[0]+1
    xArray = list(range(1,lens+1))
    # print(xArray)
    thresHold = 0.8
    # print(sortedFeatValue)
    sums = np.sum(sortedFeatValue)
    pos = 1
    tempSum = sortedFeatValue[0]

    for ele in sortedFeatValue:
        temp = tempSum / sums
        if temp > thresHold:

            break
        else:
            tempSum += ele
            pos += 1

    xMax = lens + 1
    yMax = maxValue
    return pos,xMax,yMax,xArray,sortedFeatValue
```

累计方差百分比图如下：



用选出来的最优的 k 个特征值(按从大到小排序)选择特征向量

```python
#选取前k个特征值(对特征值按由大到小排序)所代表的特征向量
def choosseFeatureVector(dataSetDiag,featVectors,k):
    # print("特征值:",dataSetDiag)
    # print("特征向量:",featVectors)
    featValue = dict()
    sortedFeatValue = []
    posList = []
    featureMatrix = []

    for i in range(dataSetDiag.shape[0]):
        featValue[dataSetDiag[i,i]] = i
    sortedFeatValue = sorted(featValue,reverse=True)

    # print(featValue)
    # print(sortedFeatValue)

    for ele in sortedFeatValue:
        print(featValue.get(ele))
        posList.append(featValue.get(ele))

    #由大到小排好序之后的位置列表
    # print(posList)


    for i in range(k):
        featureMatrix.append(featVectors[posList[i]])   #?
    # print(np.mat(featureMatrix))

    # 返回特征向量
    return np.mat(featureMatrix)
```

4.算法执行结果(用 k-means 算法进行测试)

```
C:\Users\Administrator\Anaconda3\python.exe D:/pythonworkspace/com/sun/tong/mlAlgrithom/PCA/PCA_
最佳特征的个数为：2
PCA降维前算法执行时间：0.016257565910321962
PCA降维后算法执行时间：0.012313906533775624


最终降维后的数据 [[ 0.38704229  0.38704229 -2.5157749  -1.35464803  1.54816917 -0.77408459
   2.70929605 -1.93521146  1.54816917]
 [ 0.90843217  0.16516949  0.90843217 -2.80788127  2.39495755  1.65169486
  -3.55114396  1.65169486 -1.32135589]]

Process finished with exit code 0
```

可以看到，降维后算法执行时间明显降低

5.总结

PCA 的实现过程中，由于角度的计算有点误差，导致当矩阵的阶数太高的时候特征值求出来稍微有点误差，这个还有待解决，并且由于数据集是随机生成的，导致降维之后的特征不是特别好。总的来说，还是用雅克比算法简单的实现了矩阵求特征值和特征向量。

2.基于 J48 的特征选择算法实现。

1.概念

决策树（decision tree）是一个树结构（可以是二叉树或非二叉树）。其每个非叶节点表示一个特征属性上的测试，每个分支代表这个特征属性在某个值域上的输出，而每个叶节点存放一个类别。使用决策树进行决策的过程就是从根节点开始，测试待分类项中相应的特征属性，并按照其值选择输出分支，直至到达叶子节点，将叶子节点存放的类别作为决策结果。 决策树的构造就是进行属性选择度量确定各个特征属性之间的拓扑结构。 构造决策树的关键步骤是分裂属性。所谓分裂属性就是在某个节点处按照某一特征属性的不同划分构造不同的分支，其目标是让各个分裂子集尽可能地“纯” 。尽可能“纯” 就是尽量让一个分裂子集中待分类项属于同一类别。当对属性进行分类时要判断属性属于连续值还是离散值，因为决策树只能对离散属性进行处理，所以就必须对数据进行预处理，对数据进行离散化或者装箱等的操作。

2.算法原理

a.选择属性的度量标准

(1) 信息熵

$$info(D) = -\sum_{i=1}^{m} p_i log_2(p_i)$$

其中 pi 表示第 i 个类别在整个训练数据集中出现的概率，用属于此类别元素的数量除以训练元组元素总数量作为估计。熵的实际意义表示是 D 中元组的类标号信息量的均值。

(2) 信息增益

现在我们假设将训练元组 D 按属性 A 进行划分，则 A 对 D 划分的期望信息为：

$$info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} info(D_j)$$

则信息增益为:

$$gain(A) = info(D) - info_A(D)$$

(3) 信息增益率

属性分裂时的信息为：

$$split\_info_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} log_2(\frac{|D_j|}{|D|})$$

增益率:

$$gain\_ratio(A) = \frac{gain(A)}{split\_info(A)}$$

3.算法源码

本地数据集的加载，如下图：

```python
#读取训练集
def loadDataSet():
    f = open("D://数据处理数据源//j48_train.txt",'r')
    dataSet = f.readlines()
    lables = []
    dataMat = []
    attrValue = []
    for line in dataSet:
        line = line.strip().split(",")
        lables.append(line[-1])
        dataMat.append(line[0:(len(line))])
    attrValue = dataMat[0]
    dataMat.pop(0)
    lables.pop(0)
    attrValue.pop(-1)
    return lables,attrValue,dataMat
```

递归的建立决策树 如下图

```python
#创建树
def createTree(dataSet,attrValue,attrList,N,lens):

    #数据集类标记信息的集合 ['是', '是', '是', '是', '是', '否', '否', '否', '否', '否']
    classArray = [data[-1] for data in dataSet]
    if classArray.count(classArray[0]) == len(classArray):
        return classArray[0]

    if len(dataSet[0]) == 1:
        return calVoteCount(classArray)

    #bestFeat返回字段在所在行的位置
    bestFeatPos = getBestSingleEntryDataSet(dataSet)

    bestFeatLabel = attrValue[bestFeatPos]
    # print("最佳位置:", bestFeatLabel)

    attrList.append(bestFeatLabel)


    myTree = {bestFeatLabel:{}}
    subLabels = attrValue[:]
    subLabels.pop(bestFeatPos)
    # print("层级关系:",lens-len(subLabels))
    featValues = [data[bestFeatPos] for data in dataSet]
    noRepeatValue = set(featValues)

    #如果当前树的层次等于给定的N值，则返回当前所建好的树
    if lens-len(subLabels) == N:
        return myTree

    for data in noRepeatValue:
        #程序递归生成子数
        myTree[bestFeatLabel][data] = createTree(getSubDataSet(dataSet, bestFeatPos, data), subLabels,attrList,N,lens)
```

因为要选择决策树的第 N 层以上的属性作为属性子集，所以要寻找最佳 N 值
首先计算决策树的深度，然后 N 从 2 开始一直到决策树的深度值 依次循环，每次都要计算当前 N 值的 F 值，循环完成之后，寻找最好 F 值对应的 N 值，作为最佳 N 值，然后利用最

佳 N 值，将特征子集挑选出来，然后用朴素贝叶斯分类器测试降维前后的效果

最佳 N 值的选择如下：

```
#找到最佳N值的函数(N值表示决策树的层次)
def getBestN(FList, attrSubList, attrValue, dataMat, labels, lenAttrValue, testDataSet):

    # 假设选择最上面N(假设N的范围是从2到5)层节点所用到的特征作为特征子集。必须对N有一个评价
    for N in range(2, 5):
        myTree = createTree(dataMat, attrValue, attrSubList, N, lenAttrValue)
        attrSubList.clear()
        f = getEvaluateTarget(labels, myTree, testDataSet)
        FList[N] = f
    maxValue = 0
    bestN = 0
    for k, v in FList.items():
        if v > maxValue:
            maxValue = v
            bestN = k
    # print(bestN)
    # print(FList)
    return bestN
```

在用最佳 N 值建立决策树的过程中，若当前决策树的深度等于 N,则返回建立好的树

Lens 为特征的总长度，len(subLabels)为当前决策树所在层次的特征的长度

```
#如果当前树的层次等于给定的N值，则返回当前所建好的树
if lens-len(subLabels) == N:
    return myTree
```

利用选择出来的最佳 N 值以及特征子集计算数据集

```
#求解降维之后的子数据集
def getChoosedDataSet(dataSet,attrValue,attrSubList):
    # print(dataSet)
    posList = []
    subDataSet = []
    for ele in attrSubList:
        posList.append(attrValue.index(ele))

    # print(posList)
    dataMat = np.mat(dataSet)
    for pos in posList:
        subDataSet.append(dataMat[:,pos])
    subDataSet.append(dataMat[:,-1])

    #根据选出来的特征子集选择subDataSet
    subDataMat = np.hstack(subDataSet)
    return subDataMat,dataMat
```

利用朴素贝叶斯分类器测试(在测试之前，必须对数据集进行数值化)降维前后的效果

数据集的数值化，如下

```
#矩阵由字符类型转换成数值类型(便于在分类器上边进行测试)
def string2numeric(dataMat):
    row,col = dataMat.shape
    newMat = np.zeros((row,col),dtype=int)
    # print(newMat)
    # print(dataMat)
    for i in range(col):
        # print(dataMat[:,i])
        tempValue = list(removeRepeatValue(dataMat[:,i].flatten().A[0]))

        lens = len(tempValue)
        # print(tempValue)
        for j in range(row):
            if len(tempValue) == 2:
                if dataMat[j,i] == tempValue[0]:
                    newMat[j,i] = int(6)
                if dataMat[j,i] == tempValue[1]:
                    newMat[j,i] = int(8)
            else:
                if dataMat[j,i] == tempValue[0]:
                    newMat[j,i] = int(1)
                if dataMat[j,i] == tempValue[1]:
                    newMat[j,i] = int(2)
                if dataMat[j,i] == tempValue[2]:
                    newMat[j,i] = int(3)
                pass
    # print(newMat)
    return newMat
```

去除矩阵中每一列的重复值，如下

```
#去除矩阵中元素重复数值
def removeRepeatValue(data):
    newData = []
    for ele in data:
        if ele not in newData:
            newData.append(ele)
    return newData
```

利用分类器对降维前后的数据集合进行测试，如下

```
#将特征集合提取出来之后在分类器上边进行测试
def testPerPerformance(dataMat):
    row,col = dataMat.shape

    gnb = NB.GaussianNB()

    # 朴素贝叶斯预测数值
    preditValue = gnb.fit(dataMat[:,0:col-1], dataMat[:,-1]).predict(dataMat[:,0:col-1])

    # 样本总数
    sampleSums = row

    # 预测错误的个数
    falsePreditNums = (dataMat[:,-1] != preditValue).sum()
    precision = (sampleSums - falsePreditNums) / sampleSums

    #五折交叉验证
    scores = cross_val_score(gnb, dataMat[:,0:col-1],dataMat[:,-1], cv=5)
    # print("分数为:",scores)
    return np.mean(scores)
```

```
data = string2numeric(dataMats)
start1 = t.clock()
precison = testPerPerformance(data)
end1 = t.clock()
print("降维前算法执行时间:",end1 - start1)
print("降维前算法预测准确率",precison)

print("\n")

#降维后

data = string2numeric(subDataMat)
start = t.clock()
precison = testPerPerformance(data)
end = t.clock()
print("降维后算法执行时间",end - start)
print("降维后算法预测准确率", precison)
```

4.算法执行结果

```
选择出来的属性子集为: ['色泽', '敲声', '根蒂', '纹理']


降维前算法执行时间: 0.0114071681469945
降维前算法预测准确率 0.5


降维后算法执行时间 0.010861282462827374
降维后算法预测准确率 0.6
```

5.总结

在用选择好的 N 值对决策树进行返回的时候，必须对砍掉的叶子结点进行投票，以便于确定他的类别标记，若不进行投票，则最终的衡量值精度不是特别准确,在用贝叶斯分类器进行测试的时候，需要对数据集进行数值化，以便于计算。