目录

1、概念理解

1.1、机器学习基本概念理解

编号	色泽	根蒂	含糖率	好瓜
1	青绿	蜷缩	0.46	是
2	乌黑	蜷缩	0.376	是
3	青绿	硬挺	0.268	否
4	乌黑	稍卷	0.091	否

数据集：这组记录的集合称为一个“数据集”。

样本(sample)：其中每条记录是关于一个事件或对象（这里是一个西瓜）的描述，称为一个“示例”(instance)或“样本”(sample)。

标签：又称标记，是训练样本的“结果“，如上表的“好瓜”。

属性(attribute)：反映事件或对象在某方面的表现或性质的事项，称为属性(attribute)或特征(feature)。属性上的取值称为“属性值”(attribute value)。

连续属性：属性值为一系列连续的数值，如上表含糖率。

离散属性：属性值为离散值，如上表色泽，根蒂。

属性空间：由属性张成的空间称为“属性空间”(attribute space)、“样本空间”(sample space)或“输入空间”。

特征向量：样本在属性空间中所对应的向量。

训练集：从数据中学得模型的训练过程所使用的数据的集合。

训练样本：训练集中的样本。

测试集：使用模型进行预测时被预测的样本的集合。

分类：预测过程所有预测的标记是离散值的学习任务称为分类。如上表西瓜的分类。

二分类：只有两个类别，如上表的西瓜分类。两个类分别称为正类和反类。

聚类：将训练集的西瓜分为若干组，每一组称为一个“簇”(cluster)，这些自动形成的簇可能对应一些潜在的概念划分，如“浅色瓜”，“深色瓜”，学习过程中所使用的样本通常不含标记信息。

回归：预测的是连续的值，可以用直线或者曲线来拟合。如，西瓜的成熟度。

泛化能力：即抵抗过拟合的能力，学得的模型适用于新样本的能力。

监督学习：即训练时，训练样本中含有标签，知道分类的结果。通过答案反向解题。

非监督学习：训练样本中不含有标签，不知道分类的结果。

标准化：将数据比例缩放，使之落入一个小的待定区域，一般是[-1,1]，常用的标准化方法有 min-max 和 z-score。

1.2、数据分析概念理解

数据分析是统计学的产物，顾名思义，就是分析数据。数据分析可帮助人们作出判断，以便采取适当行动。在如今这个大数据环境下，对数据分析靠人力是不现实的，使用计算机通过机器学习我想是一个分析数据的很好的办法。

2、算法理解

2.1、聚类方法

聚类分析以相似性为基础，在一个聚类中的模式之间比不在同一聚类中的模式之间具有更多的相似性。

2.1.1、K-means

描述： k-means 是划分方法中较经典的聚类算法之一。由于该算法的效率高，所以在对大规模数据进行聚类时被广泛应用。目前，许多算法均围绕着该算法进行扩展和改进。

k-means 算法目标是，以 k 为参数，把 n 个对象分成 k 个簇，使簇内具有较高的相似度，而簇间的相似度较低。

k-means 算法的处理过程如下：首先，随机地 选择 k 个对象，每个对象初始地代表了一个簇的平均值或中心;对剩余的每个对象，根据其与各簇中心的距离，将它赋给最近的簇;然后重新计算每个簇的平均值。 这个过程不断重复，直到准则函数收敛。通常，采用平方误差准则，其定义如下：

$$E\left(E=\sum_{i=1}^{k}\sum_{x\in ci}||x-ui||2\right)$$

这里 E 是数据库中所有对象的平方误差的总和，x 是空间中的点，ui 是簇 Ci 的平均值。该目标函数使生成的簇尽可能紧凑独立，使用的距离度量是欧几里得距离,当然也可以用其他距离度量。k-means 聚类算法的算法流程如下：

输入：包含 n 个对象的数据库和簇的数目 k；

输出：k 个簇，使平方误差准则最小。

步骤：

(1) 任意选择 k 个对象作为初始的簇中心；

(2) repeat；

(3) 根据簇中对象的平均值，将每个对象(重新)赋予最类似的簇；

(4) 更新簇的平均值，即计算每个簇中对象的平均值；

(5) until 不再发生变化。

总结：

优点：简单直接（体现在逻辑思路以及实现难度上），易于理解，在低维数据集上有不错的效果。

缺点：对于高维数据（如成百上千维，现实中还不止这么多），其计算速度十分慢，主要是慢在计算距离上（参考欧几里得距离，当然并行化处理是可以的，这是算法实现层面的问题），它的另外一个缺点就是它需要我们设定希望得到的聚类数 k，若我们对于数据没有很好的理解，那么设置 k 值就成了一种估计性的工作。

2.2、分类器

分类是数据挖掘的一种非常重要的方法。分类的概念是在已有数据的基础上学会一个分类函数或构造出一个分类模型（即我们通常所说的分类器(Classifier)）。该函数或模型能够把数据库中的数据纪录映射到给定类别中的某一个，从而可以应用于数据预测。总之，分类器是数据挖掘中对样本进行分类的方法的统称，包含决策树、逻辑回归、朴素贝叶斯、神经网络等算法。

2.2.1、C4.5 决策树

简介：C4.5 算法是用于生成决策树的一种经典算法，是 ID3 算法的一种延伸和优化。C4.5 算法对 ID3 算法主要做了一下几点改进：

（1）通过信息增益率选择分裂属性，克服了 ID3 算法中通过信息增益倾向于选择拥有多个属性值的属性作为分裂属性的不足；

（2）能够处理离散型和连续型的属性类型，即将连续型的属性进行离散化处理；

（3）避免过拟合，构造决策树之后进行剪枝操作；

（4）能够处理具有缺失属性值的训练数据。

分裂属性选择：分裂属性选择的评判标准是决策树算法之间的根本区别。区别于 ID3 算法通过信息增益选择分裂属性，C4.5 算法通过信息增益率选择分裂属性。

信息熵：

是度量样本集合纯度最常用的一种指标，假定当前样本集合 D 中第 k 类样本所占比例为 p_k（k = 1, 2, ...,|y| ），则 D 的信息熵定义为：

$$Ent(\mathrm{D}) = \sum_{k=1}^{|\mathrm{y}|} p_x \log_2 p_x$$

信息增益：

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

假定离散属性 a 有 V 个可能的取值，若使用 a 来对样本集 D 进行划分，则会产生 V 个分支结点，其中第 v 个分支结点包含了 D 中所有在属性 a 上取值为

a^v的样本，记为D^v。

信息增益率准则对可取值数目较多的属性有所偏好，所以提出增益率(gain ratio)

$$Gain\_ratio(D,a) = \frac{Gain(D,a)}{IV(a)}$$

其中

$$\mathrm{IV(a)} = -\sum_{v=1}^{V}\frac{|D^v|}{|D|}\log_2\frac{|D^v|}{|D|}$$

称为属性 a 的“固有值”。属性 a 的可能性取值数目越多（即 V 越大），则 IV(a)的值通常会增大。但增益率准则对可取值数目较少的属性有所偏好，因此，C4.5 算法使用启发式：先从候选划分属性中找出信息增益高于平均水平的属性，再从中选择增益率最高的。

Tips：连续属性与离散属性的信息增益计算公式有所不同。

2.2.2、KNN

KNN 算法又称 k 近邻分类(k-nearest neighbor classification)算法。它是根据不同特征值之间的距离来进行分类的一种简单的机器学习方法，它是一种简单但是懒惰的算法。他的训练数据都是有标签的数据，即训练的数据都有自己的类别。KNN 算法主要应用领域是对未知事物进行分类，即判断未知事物属于哪一类，判断思想是，基于欧几里得定理，判断未知事物的特征和哪一类已知事物的的特征最接近。它也可以用于回归，通过找出一个样本的 k 个最近邻居，将这些邻居的属性的平均值赋给该样本，就可以得到该样本的属性。K 的取值直接影响最后的结果。

2.2.3、朴素贝叶斯

基于概率论的分类算法，原理简单，多用于文本分类。利用贝叶斯公式：

$$\mathrm{P}(B_i|\mathrm{A}) = \frac{P(B_i)P(A|B_i)}{\sum_{j=1}^{n} P(B_j)P(A|B_j)}$$

假设基础：对于每一个特征都有独立、相等，导致难以找到合适的数据集。

缺点：需要知道先验概率，且先验概率很多时候取决于假设，假设的模型可以有很多种，因此在某些时候会由于假设的先验模型的原因导致预测效果不佳。

2.3、统计学习

2.3.1、SVM

SVM 的分类思想本质上和线性回归 LR 分类方法类似，就是求出一组权重系数，在线性表示之后可以分类。先使用一组 trainging set 来训练 SVM 中的权重系数，然后可以对 testingset 进行分类。

2.4、集成学习

集成学习(ensemble learning)，它本身不是一个单独的机器学习算法，而是通过构建并结合多个机器学习器来完成学习任务。集成学习有两个主要的问题需要解决，第一是如何得到若干个个体学习器，第二是如何选择一种结合策略，将这些个体学习器集合成一个强学习器。

个体学习器有两种，第一种就是所有的个体学习器都是一个种类的，或者说是同质的。

第二种是所有的个体学习器不全是一个种类的，或者说是异质的。

同质个体学习器按照个体学习器之间是否存在依赖关系可以分为两类：

第一个是个体学习器之间存在强依赖关系，一系列个体学习器基本都需要串行生成，代表算法是 boosting 系列算法。

第二个是个体学习器之间不存在强依赖关系，一系列个体学习器可以并行生成，代表算法是 bagging 和随机森林（Random Forest）系列算法。

2.4.1、AdaBoost

Boosting，也称为增强学习或提升法，是一种重要的集成学习技术，能够将预测精度仅比随机猜度略高的弱学习器增强为预测精度高的强学习器，这在直接构造强学习器非常困难的情况下，为学习算法的设计提供了一种有效的新思路和新方法。其中最为成功应用的是，Yoav Freund 和 Robert Schapire 在 1995 年提出的 AdaBoost 算法。

AdaBoost 是英文"Adaptive Boosting"（自适应增强）的缩写，它的自适应在于：前一个基本分类器被错误分类的样本的权值会增大，而正确分类的样本的权值会减小，并再次用来训练下一个基本分类器。同时，在每一轮迭代中，加入一个新的弱分类器，直到达到某个预定的足够小的错误率或达到预先指定的最大迭代次数才确定最终的强分类器。

Adaboost 算法可以简述为三个步骤：

（1）首先，是初始化训练数据的权值分布 D1。假设有 N 个训练样本数据，则每一个训练样本最开始时，都被赋予相同的权值：w1=1/N。

（2）然后，训练弱分类器 hi。具体训练过程中是：如果某个训练样本点，被弱分类器 hi 准确地分类，那么在构造下一个训练集中，它对应的权值要减小；相

反，如果某个训练样本点被错误分类，那么它的权值就应该增大。权值更新过的样本集被用于训练下一个分类器，整个训练过程如此迭代地进行下去。

（3）最后，将各个训练得到的弱分类器组合成一个强分类器。各个弱分类器的训练过程结束后，加大分类误差率小的弱分类器的权重，使其在最终的分类函数中起着较大的决定作用，而降低分类误差率大的弱分类器的权重，使其在最终的分类函数中起着较小的决定作用。

也就是说，误差率低的弱分类器在最终分类器中占的权重较大，否则较小。

2.5、线性模型

线性回归是一种监督学习下的线性模型，线性回归试图从给定数据集中学习一个线性模型来较好的预测输出。其中只需要学得 w 和 b 就可以了。使用最小二乘法求解 w 和 b，即基于均方误差最小化来进行模型求解。

3、算法实现

说明：代码实现均使用的 UCI 上数据集。采用 k 折交叉验证，验证自己算法模型的稳定性。由于 K 折交叉验证是最后才使用的，所以 PDF 里的截图可能没有更新，不过在具体的代码文件里均已更新。

3.1、KNN 算法

3.1.1、代码实现：

分类:

```python
import numpy as np
import random


def deal_with_res(k):
    lines_set = open('E:\myCodes\python\mlTest\\venv\src\d'
                     'ata\iris.data.txt').readlines()
    data = []
    for temp in lines_set:
        temp2 = list(temp[:].split("\n")[0].split(","))
        tmp = []
        for i in range(len(temp2)):
            if i != len(temp2) - 1:
                tmp.append(float(temp2[i]))
            else:
                tmp.append(temp2[i])
        data.append(tmp[:])
    list1 = list(range(0, len(data), k))
    w = random.sample(list1, 1)[0]
    data_x = []
    data_x.extend(data[:w])
    data_x.extend(data[w + int(len(data) / k):])
    data_y = [i[-1:] for i in data_x]
    data_x = [i[:-1] for i in data_x]
    test_y = [i[-1:] for i in data[w:w + int(len(data) / k)]]
    test_x = [i[:-1] for i in data[w:w + int(len(data) / k)]]
    return data_x, data_y, test_x, test_y
```

```python
    temp = [list_sub(temp, inputs) for temp in test_x]
    temp2 = np.sum(temp, axis=1)
    distance = np.ndarray.tolist(temp2 ** 0.5)
    # 返回排序后的索引值
    sorted_dist_indices = arg_sort(distance)
    class_count = {}
    # 选择k个最近邻
    for k in range(k):
        vote_labels = test_y[sorted_dist_indices[k][0]]
        temp = str(vote_labels)[2:-2]
        class_count[temp] = class_count.get(temp, 0) + 1
    max_count = 0
    # 找出出现次数最多的类别
```

```
    for k in class_count:
        if class_count.get(k) > max_count:
            max_count = class_count.get(k)
            max_key = k
    return max_key


def list_sub(a, b):
    a = list(map(float, a))
    b = list(map(float, b))
    return list(np.fabs(np.array(a) - np.array(b)))


def calc(test_x, test_y, inputs, k):
    # 计算距离
    temp = [list_sub(temp, inputs) for temp in test_x]
    temp2 = np.sum(temp, axis=1)
    distance = np.ndarray.tolist(temp2 ** 0.5)
    # 返回排序后的索引值
    sorted_dist_indices = arg_sort(distance)
    class_count = {}
    # 选择k个最近邻
    for a in range(k):
        vote_labels = test_y[sorted_dist_indices[a][0]]
        temp = str(vote_labels)[2:-2]
        class_count[temp] = class_count.get(temp, 0) + 1

    max_count = 0
    for c in class_count:
        if class_count.get(c) > max_count:
            max_count = class_count.get(c)
            max_key = c

    return max_key


def arg_sort(distance):
    return sorted(enumerate(distance), key=lambda x: x[1])
```

```python
if __name__ == '__main__':
    # list1[0] list1[1] list1[2] list1[3]
    # 分别代表样本属性，样本标签，测试属性，测试标签
    p = 10
    K = 5
    avg_accuracy = 0
    for w in range(p):
        accuracy = 0
        list1 = deal_with_res(p)
        total = len(list1[2])
        count = 0
        for i in range(total):
            pridict = calc(list1[0], list1[1], list1[2][i], K)
            if pridict == "".join(list1[3][i]):
                count = count + 1
        lebal_count = set(j[0] for j in list1[1])
        accuracy = count / total * 100
        avg_accuracy += accuracy / p
        print("第", (w + 1), "次测试准确率为：", accuracy, "%")
        if w == p-1:
            print("样本数量为：", len(list1[0]))
            print("测试数量为：", len(list1[2]))
            print("属性类为：", len(list1[0][0]))
            print("标签结果：", len(lebal_count))
            print("K值为：", K)
            print("测试平均准确率为：", avg_accuracy, "%")
```

数据集一：Iris

```
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
6.6,2.9,4.6,1.3,Iris-versicolor
5.2,2.7,3.9,1.4,Iris-versicolor
6.1,2.9,4.7,1.4,Iris-versicolor
4.4,2.9,1.4,0.2,Iris-setosa
6.4,3.2,5.3,2.3,Iris-virginica
6.5,3.0,5.5,1.8,Iris-virginica
6.4,3.1,5.5,1.8,Iris-virginica
```

结果：

```
第 1 次测试准确率为： 86.66666666666667 %
第 2 次测试准确率为： 100.0 %
第 3 次测试准确率为： 100.0 %
第 4 次测试准确率为： 86.66666666666667 %
第 5 次测试准确率为： 100.0 %
第 6 次测试准确率为： 90.0 %
第 7 次测试准确率为： 100.0 %
第 8 次测试准确率为： 86.66666666666667 %
第 9 次测试准确率为： 93.33333333333333 %
第 10 次测试准确率为： 100.0 %
样本数量为： 135
测试数量为： 15
属性类为： 4
标签结果： 3
K值为： 5
测试准确率为： 94.33333333333334 %
```

分析：

使用 K 折交叉验证有效避免了过拟合。

数据集二：letter-recognition

```
1   0,3,4,4,3,2,8,7,7,5,7,6,8,2,8,3,8
2   C,7,10,5,5,2,6,8,6,8,11,7,11,2,8,5,9
3   T,6,11,6,8,5,6,11,5,6,11,9,4,3,12,2,4
4   J,2,2,3,3,1,10,6,3,6,12,4,9,0,7,1,7
5   J,1,3,2,2,1,8,8,2,5,14,5,8,0,7,0,7
6   H,4,5,5,4,4,7,7,6,6,7,6,8,3,8,3,8
7   D,3,1,4,2,2,7,7,6,7,6,5,5,2,8,3,7
8   I,0,3,0,1,0,7,7,1,7,7,6,8,0,8,2,8
9   H,2,3,4,2,2,8,7,3,5,10,6,8,3,8,2,8
10  T,9,11,9,8,7,7,10,2,9,11,9,5,4,11,5,5
11  Y,7,9,7,7,4,3,10,2,7,11,11,7,1,11,2,5
```

结果：

```
第 1 次测试准确率为： 82.7485380116959 %
第 2 次测试准确率为： 85.08771929824562 %
第 3 次测试准确率为： 87.71929824561403 %
第 4 次测试准确率为： 87.13450292397661 %
第 5 次测试准确率为： 92.10526315789474 %
第 6 次测试准确率为： 89.18128654970761 %
第 7 次测试准确率为： 85.96491228070175 %
第 8 次测试准确率为： 88.30409356725146 %
第 9 次测试准确率为： 86.4406779661017 %
第 10 次测试准确率为： 86.8421052631579 %
样本数量为： 3084
测试数量为： 342
属性类为： 16
标签结果： 26
K值为： 5
测试准确率为： 87.15283972643473 %
```

分析：

对于这个数据集，结果比较理想，毕竟特征比上一个数据集多不少。

数据集三：cmc

```
24,2,3,3,1,1,2,3,0,1
45,1,3,10,1,1,3,4,0,1
43,2,3,7,1,1,3,4,0,1
42,3,2,9,1,1,3,3,0,1
36,3,3,8,1,1,3,2,0,1
19,4,4,0,1,1,3,3,0,1
38,2,3,6,1,1,3,2,0,1
21,3,3,1,1,0,3,2,0,1
27,2,3,3,1,1,3,4,0,1
45,1,1,8,1,1,2,2,1,1
```

结果：

```
第 1 次测试准确率为： 63.94557823129252 %
第 2 次测试准确率为： 31.97278911564626 %
第 3 次测试准确率为： 57.82312925170068 %
第 4 次测试准确率为： 37.41496598639456 %
第 5 次测试准确率为： 55.78231292517006 %
第 6 次测试准确率为： 55.78231292517006 %
第 7 次测试准确率为： 59.183673469387756 %
第 8 次测试准确率为： 63.94557823129252 %
第 9 次测试准确率为： 34.78260869565217 %
第 10 次测试准确率为： 34.26573426573427 %
样本数量为： 1330
测试数量为： 143
属性类为： 9
标签结果： 3
K值为： 4
测试准确率为： 49.48986830974409 %
```

分析：

不知道为什么特征比上一个还少，准确率却低，想试试 sklearn 里的算法，无奈总是 import 失败。初步推测是数据的原因吧。

回归：替换核心代码

```
def calc(test_x, test_y, inputs, k):
    # 计算距离
    temp = [list_sub(temp, inputs) for temp in test_x]
    temp2 = np.sum(temp, axis=1)

    distance = np.ndarray.tolist(temp2 ** 0.5)
    # 返回排序后的索引值
    sorted_dist_indices = arg_sort(distance)
    class_count = {}
    # 选择k个最近邻
    for a in range(k):
        vote_labels = test_y[sorted_dist_indices[a][0]]
        temp = str(vote_labels)[1:-1]
        class_count[temp] = class_count.get(temp, 0) + 1
    sums = 0
    for c in class_count:
        sums = sums + class_count.get(c) * float(c)
    avg = sums/k
    return avg
```

```
def arg_sort(distance):
    return sorted(enumerate(distance), key=lambda x: x[1])
```

数据集四：Winequality-red

```
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5
7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5
7.9;0.43;0.21;1.6;0.106;10;37;0.9966;3.17;0.91;9.5;5
8.5;0.49;0.11;2.3;0.084;9;67;0.9968;3.17;0.53;9.4;5
6.9;0.4;0.14;2.4;0.085;21;40;0.9968;3.43;0.63;9.7;6
6.3;0.39;0.16;1.4;0.08;11;23;0.9955;3.34;0.56;9.3;5
7.6;0.41;0.24;1.8;0.08;4;11;0.9962;3.28;0.59;9.5;5
7.9;0.43;0.21;1.6;0.106;10;37;0.9966;3.17;0.91;9.5;5
7.1;0.71;0;1.9;0.08;14;35;0.9972;3.47;0.55;9.4;5
7.8;0.645;0;2;0.082;8;16;0.9964;3.38;0.59;9.8;6
6.7;0.675;0.07;2.4;0.089;17;82;0.9958;3.35;0.54;10.1;5
```

结果：

```
99.43714821763602 %
99.62476547842401 %
99.812382739212 %
样本数量为： 1066
测试数量为： 533
属性类为： 11
K值为： 5
总cost为： 316.7999999999996
```

分析：

或许是属性有点多，该回归任务完成的不是很理想。对于含有差别较大属性的数据，需要归一化操作。

3.1.2、小结

KNN 算法

3.2、C4.5 决策树

3.2.1、代码实现

```
from scipy import *
from math import log


# 计算给定数据的香浓熵:
def calc_shannon_ent(data_set):
    num_entries = len(data_set)  # 数据总个数
    label_counts = {}  # 标签类别字典（类别的名称为键，该类别的个数为值）
    for featVec in data_set:
        current_label = featVec[-1]  # 选取最后一列，为标签列
        if current_label not in label_counts.keys():  # 还没添加到字典里的类型
            label_counts[current_label] = 0
        label_counts[current_label] += 1
    shannon_ent = 0.0
    for keys in label_counts:  # 求出每种类型的熵
        prob = float(label_counts[keys]) / num_entries  # 每种类型个数占所有的比值
        shannon_ent -= prob * log(prob, 2)
    return shannon_ent  # 返回熵


# 按照给定的特征划分数据集,剔除含选出属性的一列，lisan属性为true即为离散
def split_data_set(data_set, axis, value, flag, lisan=True):
    ret_data_set = []
    if lisan:
        # 按dataSet矩阵中的第axis列的值等于value的分数据集
        for featVec in data_set:
            # 值等于value的，每一行为新的列表（去除第axis个数据）
            if featVec[axis] == value:
                reduced_feat_vec = featVec[:axis]
                reduced_feat_vec.extend(featVec[axis + 1:])
                ret_data_set.append(reduced_feat_vec)
    else:
        for featVec in data_set:
```

```
            if flag:
                if featVec[axis] >= value:
                    reduced_feat_vec = featVec[:axis]
                    reduced_feat_vec.extend(featVec[axis + 1:])
                    ret_data_set.append(reduced_feat_vec)
            else:
                if featVec[axis] < value:
                    reduced_feat_vec = featVec[:axis]
                    reduced_feat_vec.extend(featVec[axis + 1:])
                    ret_data_set.append(reduced_feat_vec)
    return ret_data_set  # 返回分类后的新矩阵


# 选择最好的数据集划分方式
def choose_best_feature_to_split(data_set):
    num_features = len(data_set[0]) - 1  # 求属性的个数,减一是因为数据集里有结果那一列
    base_entropy = calc_shannon_ent(data_set)  # 计算 Ent(D) 计算纯度
    best_infoGain = 0.0  # 最大的信息增益
    best_feature = -1  # 选择出信息增益最大的列的下标
    for i in range(num_features):  # 求所有属性的信息增益
        feat_list = [example[i] for example in data_set]  # 获得第i列的值
```

```
        new_entropy = 0.0
        split_info = 0.0  # 属性的固有值，属性类别越多，值越大
        # 判断是否为离散属性
        if type(feat_list[0]) is float:  # 连续型
            flag = True  # 标记为连续型
            temp_list = list(set(sorted(feat_list)))  # 排序
            candidate = []  # 候选值
            # temp_list = list(set(temp_list))
            for j in range(len(temp_list) - 1):
                candidate.append((temp_list[j] + temp_list[j + 1]) / 2)  # 计算中位点
            unique_values = set(candidate)  # 所有候选值
            # 考察每一个候选值，选出最大信息熵所对应的候选值作为阈值
            infoGain = 0
            for value in unique_values:  # 求第i列属性每个不同值的熵*他们的概率
                # 获取只含小于value的样本集
                sub_data_set = split_data_set(data_set, i, value, 0, False)
                # 获取只含属性大于等于value的样本集
                sub_data_set1 = split_data_set(data_set, i, value, 1, False)
                prob_a = len(sub_data_set) / float(len(data_set))  # 求出该值在i列属性中的概率
                # 计算条件熵
                new_entropy = \
                    prob_a * calc_shannon_ent(sub_data_set) + (1 - prob_a) * calc_shannon_ent(sub_data_set1)
                sub_infoGain = (base_entropy - new_entropy)  # 求出第i列属性的信息增益
                if sub_infoGain > infoGain:  # 记录阈值信息
```

```
                    infoGain = sub_infoGain
                    if sub_infoGain > best_infoGain:
                        best_feature = [value, i]  # 我需要记录i值
        else:
            flag = False
            unique_values = set(feat_list)  # 第i列属性的取值（不同值）数集合
            for value in unique_values:  # 求第i列属性每个不同值的熵*他们的概率
                # 获取只含属性等于value的样本集
                sub_data_set = split_data_set(data_set, i, value, 0)
                # 求出该值在i列属性中的概率
                prob = len(sub_data_set) / float(len(data_set))
                # 求i列属性各值对应的熵求和
                new_entropy += prob * calc_shannon_ent(sub_data_set)
                split_info -= prob * log(prob, 2)
                # 求出第i列属性的信息增益率
            infoGain = (base_entropy - new_entropy) / split_info
            # 保存信息增益率最大的信息增益率值以及所在的下标（列下标i）
        if infoGain > best_infoGain:
            best_infoGain = infoGain
            if not flag:
                best_feature = i
    return best_feature
```

```
# 找出出现次数最多的分类名称
def majority_cnt(class_list):
    class_count = {}
    for vote in class_list:
        if vote not in class_count.keys():
            class_count[vote] = 0
        class_count[vote] += 1
    max_1 = 0
    for d in class_count:
        if max_1 < class_count.get(d):
            max_1 = class_count.get(d)
            result = d
    return result


# 创建树
def create_tree(data_set, labels_c):
    # 创建需要创建树的训练数据的结果列表
    class_list = [example[-1] for example in data_set]
    # 如果所有的训练数据都是属于一个类别，则返回该类别
    if class_list.count(class_list[0]) == len(class_list):
```

```
        return class_list[0]
    # 训练数据只给出类别数据（没给任何属性值数据），返回出现次数最多的分类名称
    if len(data_set[0]) == 1:
        return majority_cnt(class_list)
    if len(data_set) <= 3:
        return majority_cnt(class_list)
    # 选择信息增益最大的属性进行分（返回值是属性类型列表的下标，连续型为节点值和下标）
    best_feat = choose_best_feature_to_split(data_set)

    # 剪枝操作
    if type(best_feat) == list:  # 为连续属性
        best_feat_label = labels_c[best_feat[1]]
        my_tree = {best_feat_label: {}}  # 以bestFeatLabel为根节点建一个空树
        del (labels_c[best_feat[1]])  # 从属性列表中删掉已经被选出来当根节点的属性
        key1 = "<" + str(best_feat[0])
        key2 = ">=" + str(best_feat[0])
        # 这里大概花了我两天的时间，因为python list递归会出现问题，
        # 所以需要复制操作 sub_labels = labels_c[:]
        sub_labels = labels_c[:]
        # split_data_set 函数 倒数第二个参数 1代表选中大于value的 0 代表小于
        return_tree1 = \
            create_tree(split_data_set(data_set, best_feat[1], best_feat[0], 0, False), sub_labels)
        sub_labels = labels_c[:]
```

```
        return_tree2 = \
            create_tree(split_data_set(data_set, best_feat[1], best_feat[0], 1, False), sub_labels)
        my_tree[best_feat_label][key1] = return_tree1  # 根据各个分支递归创建树
        my_tree[best_feat_label][key2] = return_tree2  # 根据各个分支递归创建树
    else:
        best_feat_label = labels_c[best_feat]  # 根据下表找属性名称当树的根节点
        my_tree = {best_feat_label: {}}  # 以bestFeatLabel为根节点建一个空树
        del (labels_c[best_feat])  # 从属性列表中删掉已经被选出来当根节点的属性
        # 找出该属性所有训练数据的值（创建列表）
        feat_values = [example[best_feat] for example in data_set]
        unique_values = set(feat_values)  # 求出该属性的所有值得集合（集合的元素不能重复）
        for value in unique_values:  # 根据该属性的值求树的各个分支
            sub_labels = labels_c[:]
            # 根据各个分支递归创建树
            my_tree[best_feat_label][value] = \
                create_tree(split_data_set(data_set, best_feat, value, 0), sub_labels)
    return my_tree  # 生成的树


# 实用决策树进行分类
def classify(input_tree, feat_labels, test_vec):
```

```
    global class_label
    first_str = str(input_tree.keys())[12:-3]  # 得到第一个字典的key
    second_dict = input_tree[first_str]
    feat_index = feat_labels.index(first_str)
    global class_label
    for key1 in second_dict.keys():
        if key1.count("<"):     # 连续
            key2 = float(key1[1:])
            if test_vec[feat_index] <= key2:
                if type(second_dict[key1]).__name__ == 'dict':  # 如果没有到最底层，递归
                    class_label = classify(second_dict[key1], feat_labels, test_vec)
                else:
                    class_label = second_dict[key1]
        elif key1.count(">="):
            key2 = float(key1[2:])
            if test_vec[feat_index] > key2:
                if type(second_dict[key1]).__name__ == 'dict':  # 如果没有到最底层，递归
                    class_label = classify(second_dict[key1], feat_labels, test_vec)
                else:
                    class_label = second_dict[key1]
        else:                                   # 离散
            if test_vec[feat_index] == key1:
                if type(second_dict[key1]).__name__ == 'dict':  # 如果没有到最底层，递归
                    class_label = classify(second_dict[key1], feat_labels, test_vec)
```

```
                else:
                    class_label = second_dict[key1]
    return class_label


# 读取数据文档中的训练数据（生成二维列表）
def create_train_data():
    lines_set = open('E:\myCodes\python\mlTest\\veny\src\data\letter_train.txt').readlines()
    data_x = []
```

```
    for temp in lines_set:
        # 将标签移至最后一列
        temp2 = list(temp[:].split("\n")[0].split(","))
        # print(temp2)
        c = temp2[-1]
        temp2[-1] = temp2[0]
        temp2[0] = c
        temp2 = list(map(float, temp2[:-1])) + temp2[-1:]
        # print("##", temp2)
        data_x.append(temp2)
    label = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10', 'A11',
             'A12', 'A13', 'A14', 'A15', 'A16']
    return data_x, label


# 读取数据文档中的测试数据（生成二维列表）
def create_test_data():
    lines_set = open('E:\myCodes\python\mlTest\\ven'
                     'v\src\data\letter-test.txt').readlines()
    data_x = []
    data_y = []
    for temp in lines_set:
        temp2 = list(temp[:].split("\n")[0].split(","))
        c = temp2[-1]
```

```
            temp2[-1] = temp2[0]
            temp2[0] = c
            temp3 = list(map(float, temp2[:-1]))
            data_x.append(temp3)
            data_y.append(temp2[-1:])
    return data_x, data_y


if __name__ == '__main__':
    my_dat_x, labels = create_train_data()
    myTree = create_tree(my_dat_x, labels)
    print(myTree)
    # 重新创建一次属性列表，具体原因尚不太明白

    boot_labels = ['A1', 'A2', 'A3', 'A4', 'A5',
                   'A6', 'A7', 'A8', 'A9', 'A10', 'A11',
                   'A12', 'A13', 'A14', 'A15', 'A16']
    test_list, test_labels = create_test_data()
```

```
    labels_num = {}
    total = len(test_labels)
    pre = []
    count = 0
    for testData in test_list:
        pre.append(classify(myTree, boot_labels, testData))
    for ic in range(total):
        key = set(test_labels[ic])
        labels_num[str(key)] = 1
        if pre[ic] == str(test_labels[ic])[2:-2]:
            count += 1
    print("样本数量为：", len(my_dat_x))
    print("测试数量为：", len(test_list))
    print("属性类为：", len(my_dat_x[0])-1)
    print("标签结果：", len(labels_num))
    print("测试准确率为：", count / total * 100, "%")
```

C4.5 决策树的代码花了我很多时间，先完成对离散属性的分类，再完成离散与连续属性的结合。其中辨别离散还是连续属性看属性如果是 float 型则为连续，如果是字符串，则是离散，这就要求事先对数据集进行处理到位。

数据集一：Car Evaluation

离散属性

```
vhigh,high,3,more,big,med,unacc
vhigh,high,3,more,big,high,unacc
vhigh,high,4,2,small,low,unacc
vhigh,vhigh,2,2,small,low,unacc
vhigh,vhigh,2,2,small,med,unacc
vhigh,vhigh,2,2,small,high,unacc
vhigh,vhigh,2,2,med,low,unacc
vhigh,vhigh,2,2,med,med,unacc
vhigh,vhigh,2,2,med,high,unacc
vhigh,vhigh,2,2,big,low,unacc
vhigh,vhigh,2,2,big,med,unacc
vhigh,vhigh,2,2,big,high,unacc
```

结果：

```
第 1 次测试准确率为： 81.97674418604652 %
第 2 次测试准确率为： 68.6046511627907 %
第 3 次测试准确率为： 75.5813953488372 %
第 4 次测试准确率为： 95.34883720930233 %
第 5 次测试准确率为： 69.18604651162791 %
第 6 次测试准确率为： 80.23255813953489 %
第 7 次测试准确率为： 81.97674418604652 %
第 8 次测试准确率为： 70.93023255813954 %
第 9 次测试准确率为： 83.13953488372093 %
第 10 次测试准确率为： 64.0625 %
样本数量为： 1600
测试数量为： 128
属性类为： 6
标签结果： 4
测试平均准确率为： 77.10392441860466 %
```

分析：使用了简单的预剪枝操作，即如果样本小于等于 3 则不继续进行分类。

数据集二：letter-recognition

连续属性

```
459  O,3,4,4,3,2,8,7,7,5,7,6,8,2,8,3,8
460  C,7,10,5,5,2,6,8,6,8,11,7,11,2,8,5,9
461  T,6,11,6,8,5,6,11,5,6,11,9,4,3,12,2,4
462  J,2,2,3,3,1,10,6,3,6,12,4,9,0,7,1,7
463  J,1,3,2,2,1,8,8,2,5,14,5,8,0,7,0,7
464  H,4,5,5,4,4,7,7,6,6,7,6,8,3,8,3,8
465  S,3,2,3,3,2,8,8,7,5,7,5,7,2,8,9,8
466  O,6,11,7,8,5,7,6,9,6,7,5,9,4,8,5,5
467  J,3,6,4,4,2,6,6,4,4,14,8,12,1,6,1,6
468  C,6,11,7,8,3,7,8,7,11,4,7,14,1,7,4,8
469  M,7,11,11,8,9,3,8,4,5,10,11,10,10,9,5,7
```

结果：

```
第 1 次测试准确率为： 67.2514619883041 %
第 2 次测试准确率为： 71.05263157894737 %
第 3 次测试准确率为： 62.28070175438597 %
第 4 次测试准确率为： 67.54385964912281 %
第 5 次测试准确率为： 66.66666666666666 %
第 6 次测试准确率为： 69.00584795321637 %
第 7 次测试准确率为： 65.2046783625731 %
第 8 次测试准确率为： 66.66666666666666 %
第 9 次测试准确率为： 65.2046783625731 %
第 10 次测试准确率为： 69.2982456140351 %
样本数量为： 3084
测试数量为： 342
属性类为： 16
标签结果： 26
测试平均准确率为： 67.01754385964912 %
```

数据集三：winequality-red

连续属性

```
7.3;0.65;0;1.2;0.065;15;21;0.9946;3.39;0.47;10;7
7.8;0.58;0.02;2;0.073;9;18;0.9968;3.36;0.57;9.5;7
7.5;0.5;0.36;6.1;0.071;17;102;0.9978;3.35;0.8;10.5;5
6.7;0.58;0.08;1.8;0.097;15;65;0.9959;3.28;0.54;9.2;5
7.5;0.5;0.36;6.1;0.071;17;102;0.9978;3.35;0.8;10.5;5
5.6;0.615;0;1.6;0.089;16;59;0.9943;3.58;0.52;9.9;5
7.8;0.61;0.29;1.6;0.114;9;29;0.9974;3.26;1.56;9.1;5
8.9;0.62;0.18;3.8;0.176;52;145;0.9986;3.16;0.88;9.2;5
8.9;0.62;0.19;3.9;0.17;51;148;0.9986;3.17;0.93;9.2;5
8.5;0.28;0.56;1.8;0.092;35;103;0.9969;3.3;0.75;10.5;7
```

结果：

```
第 1 次测试准确率为： 58.0952380952381 %
第 2 次测试准确率为： 51.42857142857142 %
第 3 次测试准确率为： 44.761904761904766 %
第 4 次测试准确率为： 54.285714285714285 %
第 5 次测试准确率为： 43.39622641509434 %
第 6 次测试准确率为： 56.19047619047619 %
第 7 次测试准确率为： 37.142857142857146 %
第 8 次测试准确率为： 48.57142857142857 %
第 9 次测试准确率为： 40.0 %
第 10 次测试准确率为： 33.33333333333333 %
样本数量为： 948
测试数量为： 105
属性类为： 11
标签结果： 5
测试平均准确率为： 46.720575022461816 %
```

分析：感觉准确率有点低，但是我使用 python 的 sklearn 库，也是只有 48%说明我的算法问题不大。一时半会也找不出问题所在。

数据集四：Fertility

离散+连续属性：

属性信息：

进行分析的季节。1）冬天，2）春天，3）夏天，4）秋天。(-1，-0.33，0.33，1)

分析时的年龄。18-36（0,1）

幼稚疾病（即水痘，麻疹，腮腺炎，小儿麻痹症）1）是，2）否。(0,1） 事故或严重创伤 1）是，2）否。(0,1)

手术干预 1）是，2）否。(0,1)

去年高烧 1）不到三个月前，2）三个多月前，3）没有。(-1,0,1)

饮酒次数 1) 每天几次，每天 2 次，每周 3 次，每周几次，每周一次，每周一次，几乎永远或从不（0,1)

吸烟习惯 1）从不，2）偶尔 3）每天。(-1,0,1)

每天坐着的小时数 ene-16（0，1）

输出：诊断正常（N)，改变（O)

```
-0.33,0.69,0,1,1,0,0.8,0,0.88,N
-0.33,0.94,1,0,1,0,0.8,1,0.31,O
-0.33,0.5,1,0,0,0,1,-1,0.5,N
-0.33,0.75,0,1,1,0,1,-1,0.38,N
-0.33,0.67,1,1,0,0,0.8,-1,0.5,O
-0.33,0.67,1,0,1,0,0.8,0,0.5,N
-0.33,0.67,0,0,0,-1,0.8,-1,0.44,N
-0.33,1,1,1,1,0,0.6,-1,0.38,N
1,0.64,0,0,1,0,0.8,-1,0.25,N
1,0.61,1,0,0,0,1,-1,0.25,N
```

结果：

```
第 1 次测试准确率为： 80.0 %
第 2 次测试准确率为： 80.0 %
第 3 次测试准确率为： 80.0 %
第 4 次测试准确率为： 90.0 %
第 5 次测试准确率为： 100.0 %
第 6 次测试准确率为： 80.0 %
第 7 次测试准确率为： 80.0 %
第 8 次测试准确率为： 80.0 %
第 9 次测试准确率为： 80.0 %
第 10 次测试准确率为： 90.0 %
样本数量为： 90
测试数量为： 10
属性类为： 9
标签结果： 1
测试平均准确率为： 84.0 %
```

分析：

当离散属性里面只有一个取值时，会导致分裂属性为 1，导致代码出现除 0bug，不过已经修复。

3.2.2、小结

生成离散属性决策树代码是参考了网上的一些代码，我觉得写的很不错，在我完全读懂他的代码后，自己补全了关于连续属性的生成结点的部分，路上遇到的问题很多。

① python 在进行递归时应该使用 deepcopy 来保存 list 等变量，不然你的变量在递归中会出现问题。

② 关于对连续属性划分的修改后的信息增益，一开始理解错了，导致代码一直达不到理想的结果。但在一次午饭过后，我随意注释了一段代码，结果居然对了，而后我才慢慢发觉是自己理解错了公式。对连续属性进行划分不需要除以 IV(a)，即属性 a 的固有属性。

③ 没怎么实现预剪枝，因为我发现在递归途中获取不到已经生成的决策树。所

以无法进行预剪枝操作。对后剪枝理解不是很透彻。

3.3、聚类算法 k-means

3.3.1、代码实现

```
import numpy as np
import random


def distant_calc(centre, data_set):
    return np.sqrt(np.sum(np.power(centre - data_set, 2)))


def create_test_data():
    lines_set = open('E:\myCodes\python\mlTest\\ve'
                     'nv\src\data\iris.data.txt').readlines()
    data_x = []
    data_y = []
    for data in lines_set:
        temp = data.split("\n")[0].split(",")
        data_x.append(list(map(float, temp[:-1])))
        c = str(temp[-1:])[2:-2]
        data_y.append(c)
    return data_x, data_y


# def create_test_data():
#     lines_set = open('E:\myCodes\python\mlTest\\ve
# nv\src\data\Wholesale customers data.csv').readlines()
```

```
#     data_x = []
#     for data in lines_set:
#         temp =data.split("\n")[0].split(",")
#         data_x.append(list(map(float, temp)))
#     return data_x


# 初始化聚簇中心，选取K个
def init_centre(data_set, k):
    n = np.shape(data_set)[1]
    centre = np.mat(np.zeros((k, n)))
    # print(np.shape(data_set)[0])
    r = random.sample(list(range(np.shape(data_set)[0])), k)
    print('初始化簇中心', r)
    # 随机选四个数据
    for row in range(k):
        centre[row] = data_set[r[row]]
    return centre


def k_means(dataSet, k, distMeans =distant_calc, createCent = init_centre):
    m = np.shape(dataSet)[0]  # 获取数据个数
    # cluster_infor第一列存放该数据所属的中心点，第二列是该数据到中心点的距离
    cluster_infor = np.mat(np.zeros((m, 2)))  # 用于存放该样本属于哪类及质心距离
```

```
    # print(data_set)
    centroids = createCent(dataSet, k)  # 初始化类簇
    cluster_changed = True  # 用来判断聚类是否已经收敛
    while cluster_changed:
        cluster_changed = False
        for i in range(m):  # 把每一个数据点划分到离它最近的中心点
            min_dist = float('inf')
            min_index = -1
            for j in range(k):
                distJI = distMeans(centroids[j], np.mat(dataSet[i]))
                if distJI < min_dist:
                    min_dist = distJI
                    # 如果第i个数据点到第j个中心点更近，则将i归属为j
                    min_index = j
            if cluster_infor[i, 0] != min_index:
                cluster_changed = True  # 如果分配发生变化，则需要继续迭代
                # 并将第i个数据点的分配情况存入字典
            cluster_infor[i, :] = min_index, min_dist
```

```
        for cent in range(k):  # 重新计算中心点
            # 去第一列等于cent的所有列
            new_clust = dataSet[np.nonzero(cluster_infor[:, 0].A == cent)[0]]
            # 算出这些数据的中心点
            centroids[cent, :] = np.mean(new_clust, axis=0)
    return centroids, cluster_infor


# 使用外部指标度量聚类性能
def outer_judge(result, models):
    # 获取|SS|
    count_a = 0
    count_b = 0
    count_c = 0
    count_d = 0
    n = len(models)
    for i in range(n):
        for j in range(i+1, n):
            if result[i].tolist()[0][0] \
                    == result[j].tolist()[0][0] and models[i] == models[j]:
                count_a += 1
            elif result[i].tolist()[0][0]\
                    == result[j].tolist()[0][0] and models[i] != models[j]:
                count_b += 1
```

```
            elif result[i].tolist()[0][0] \
                    != result[j].tolist()[0][0] and models[i] == models[j]:
                count_c += 1
            else:
                count_d += 1
    JC = count_a / (count_a + count_b + count_c)
    FMI = np.sqrt((count_a / (count_a +
                              count_b)) * (count_a / (count_a + count_c)))
    Rand = 2 * (count_a + count_d) / (n * (n-1))
    print('JC系数为：', JC)
    print('FM指数为：', FMI)
    print('Rand指数为：', Rand)


def inner_judge(result, data_set, centroids):
    n = np.shape(result)[0]

    # 用于统计每个类簇中样本所对应在原始数据里的下标
    dic_cluster = {}
```

```
    for i in range(n):
        list1 = []
        item = result[i].tolist()[0][0]
        if item not in dic_cluster.keys():
            dic_cluster[item] = list1
        else:
            list1 = dic_cluster[item]
        list1.append(i)
    dic_avg = {}
    for i in dic_cluster:
        total_diatant = 0
        tmp = dic_cluster[i]
        for j in range(len(tmp)):
            for k in range(j + 1, len(tmp)):
                total_diatant += \
                    distant_calc(np.mat(data_set[tmp[j]]), np.mat(data_set[tmp[k]]))
        dic_avg[i] = (total_diatant * 2 / (n * (n - 1)))

    # 计算各个中心点距离
    num_cluster = len(dic_avg)
    total_max = 0
    for i in range(num_cluster):
        max_distant = 0
```

```
        for j in range(num_cluster):

            new_distant = (dic_avg[i] + dic_avg[j]) \
                          / distant_calc(np.mat(centroids[i]), np.mat(centroids[j]))
            if new_distant > max_distant:
                max_distant = new_distant
        total_max += max_distant
    DBI = total_max / num_cluster
    print('DB指数为：', DBI)
    return DBI


if __name__ == '__main__':
    k = 3
    # data_set = create_test_data()
    data_set, lables = create_test_data()
    centroids, result = k_means(np.mat(data_set), k)
    outer_judge(result, lables)
    # min_DBI = 1
    # for i in range(3, 15):
    #     centroids, result = k_means(np.mat(data_set), i)
    #     DBI = inner_judge(result, data_set, centroids)
    #     if DBI < min_DBI:
    #         min_DBI = DBI
```

```
157         #         k = i
158
159         print('K值为:', k)
160         print('簇坐标', centroids)
161         print('数据总数', len(data_set), '已分类数据：', np.shape(result)[0])
162         dic = {}
163         for i in result:
164             item = i.tolist()[0][0]
165             if item not in dic.keys():
166                 dic[item] = 0
167             dic[item] += 1
168         print('分类结果', dic)
169
```

数据集一：Iris

使用外部指标度量

结果:

```
初始化簇中心 [116, 47, 92]
JC系数为： 0.6998950682056663
FM指数为： 0.8235798153491702
Rand指数为： 0.8818181818181818
簇坐标 [[6.81666667 3.06333333 5.62666667 2.03      ]
 [4.98333333 3.38333333 1.46428571 0.24761905]
 [5.89591837 2.75510204 4.39183673 1.44081633]]
数据总数 121 已分类数据： 121
分类结果 {1.0: 42, 2.0: 49, 0.0: 30}
```

分析:

聚类性能度量外部指标 JC 系数、FM 指数、Rand 指数取值在[0,1],均是越大越好，说明聚类效果还是不错的。另外，在进行多次测试后发现，有一定的几率出现很大的偏差。我猜测这和初始化簇中心有关。

数据集二：winequality-red

结果：

```
初始化簇中心 [143, 541, 860, 817, 118, 476]
JC系数为： 0.1618039108222905
FM指数为： 0.29612237485791715
Rand指数为： 0.6134293111479423
K值为: 6
特征数： 6
数据总数 1053 已分类数据： 1053
分类结果 {1.0: 189, 0.0: 208, 4.0: 244, 3.0: 106, 2.0: 137, 5.0: 169}
```

分析：

效果没有上一个数据集好，特别是 JC 系数太小。同一个数据集，之前使用 c4.5 决策树的时候也发现这个数据集有点问题，现在可以下定论，这个数据集确实存在一些问题。

数据集三：seeds

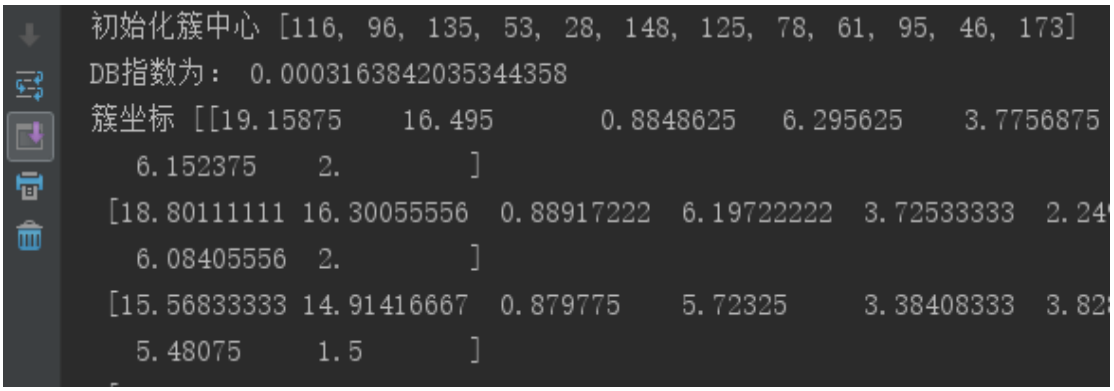
使用内部指标度量

结果：

K 取值为 3：

```
初始化簇中心 [108, 33, 44]
DB指数为： 0.022658957675026903
簇坐标 [[18.72180328 16.29737705  0.88508689  6.20893443  3.72267213  3.60
   6.06609836  1.98360656]
 [11.90906667 13.25026667  0.85154933  5.22233333  2.86509333  4.72218667
   5.09304     2.86666667]
 [14.63202703 14.45324324  0.8790973   5.56178378  3.27489189  2.74404324
   5.18493243  1.13513514]]
数据总数 210 已分类数据： 210
分类结果 {2.0: 74, 1.0: 75, 0.0: 61}
```

K 取值为 12：



分析：

采用了聚类性能度量内部指标 DBI 来度量结果，可见，取不同的 k 对结果的影响相当大，一般的 k 值在 3 到 10，可以采用逐个尝试的方式来确定 k 值。随着 K 值的增加，内部指标 DBI 肯定是越来越小，这里我们需要设定一个阈值，直到 DBI 小于阈值，我们就认为该聚类算法已经完成任务。刚开始对 DBI 的理解出了点偏差，试了很多 k 值，都是取最大 k 值时 DBI 最小，当时有点想不通，以为是代码的问题，闹了个笑话。

3.3.2、小结

① 在初始化簇中心生成随机数时，一开始使用的 random.randint()函数出现了重复数据，后面改用 random.sample()函数解决了这个问题。

② 使用内部度量指标应该设定阈值。

3.4、朴素贝叶斯分类器

3.4.1、代码实现

```
import copy
from math import log
import random


# 定义一个标签类，用于存储标签信息
class Lable(object):
    def setValue(self, value):
        self.value = value

    def setIdx_list(self, idx_list):
        self.idx_list = idx_list

    def setCount(self, count):
        self.count = count


# 读取数据文档中的训练数据（生成二维列表）
def load_train_data(k):
    lines_set = open('E:\myCodes\python\mlTest\\venv\src\d'
                     'ata\PhishingData.arff.txt').readlines()
    data = []
    for temp in lines_set:
        temp2 = list(temp[:].split("\n")[0].split(","))
        data.append(temp2[:])
    list1 = list(range(0, len(data), k))
    w = random.sample(list1, 1)[0]
    data_x = []
```

```
    data_x.extend(data[:w])
    data_x.extend(data[w + int(len(data) / k):])
    data_y = [str(i[-1:])[1:-1] for i in data_x]
    data_x = [i[:-1] for i in data_x]
    test_y = [str(i[-1:])[1:-1] for i in data[w:w +
                                              int(len(data) / k)]]
    test_x = [i[:-1] for i in data[w:w + int(len(data) / k)]]
    return data_x, data_y, test_x, test_y

```

```
def init_table(data_set, lables):
    n = len(data_set)
    lab_list = []
    lab_set = set(lables)
    # 向标签类存数据
    for lab in lab_set:
        new_lab = Lable()
        new_lab.setValue(lab)
        count = 0
        sub_lab_list = []
        for index_lab in range(n):
            if lables[index_lab] == lab:
                count += 1
                sub_lab_list.append(index_lab)
        new_lab.setCount(count)
        new_lab.setIdx_list(sub_lab_list)
        lab_list.append(new_lab)
    list_resu = []
    # 遍历每个属性
    for i in range(len(data_set[0])):
        attra_i = []  # 存放所有样本在第i个属性的所有值
        # 遍历某个属性的所有样本
        for j in range(n):
            attra_i.append(data_set[j][i])  # j行，i列
        unique_att = set(attra_i)
        dic_att = {}

        for att in unique_att:
            dic_value = {}  # 存放属性值att所对应所有标签的样本个数

            # 遍历每个标签
            for item in lab_list:
                lable = item.value  # value即是标签
                for inner in item.idx_list:
                    if data_set[inner][i] == att:
                        if lable not in dic_value.keys():
                            dic_value[lable] = 0
                        dic_value[lable] += 1
```

```
                # 根据计数结果计算概率
                if lable in dic_value.keys():
                    dic_value[lable] = \
                     (dic_value[lable] + 1) / (item.count + len(unique_att))
            # 将未出现的属性处理
            for la in lab_set:

                if la not in dic_value.keys():
                    dic_value[la] = 1 / len(data_set)
            dic_att[att] = copy.deepcopy(dic_value)
        list_resu.append(copy.deepcopy(dic_att))
    return list_resu


def native_bayes(test_data, table_prob):
    n = len(test_data)
    list2 = []  # 存放结果
    list_tmp = []
    list_prob = []  # 存放每一个样本对应每个标签的概率的集合
    for item in range(n):
        # 第i个属性， 逐个计算每一个属性对应的概率
        for i in range(len(test_data[0])):
            value = test_data[item][i]  # 代表第item行，i列的值
            list_prob.append(table_prob[i][value])
        dic_line = {}
        for sub_pro in list_prob:
            for ree in sub_pro:
                if ree not in dic_line.keys():
                    dic_line[ree] = 1
                tmp = log(sub_pro[ree], 2)
                dic_line[ree] += tmp
        list_tmp.append(dic_line)
    for item in list_tmp:
        max_prob = float('-inf')
        max_key = ''
        for dic_item in item:
            if max_prob < item[dic_item]:
                max_prob = item[dic_item]
                max_key = dic_item
        list2.append(max_key)
    return list2


if __name__ == '__main__':
    K = 10
    avg_accuracy = 0
    for i in range(K):
        count = 0
        data_set, lables, test_set, test_lables =\
            load_train_data(K)
        labels_num = set(lables)
        total = len(test_lables)
```

```
            table_prob = init_table(data_set, lables)
            predict = native_bayes(test_set, table_prob)
            for j in range(len(test_set)):
                if predict[j] == test_lables[j]:
                    count += 1
            accuracy = count / total * 100
            print("第", (i + 1), "次测试准确率为：", accuracy, "%")
            avg_accuracy += accuracy / K
            if i == K-1:
                print("样本数量为：", len(data_set))
                print("测试数量为：", len(test_set))
                print("属性类为：", len(data_set[0]))
                print("标签结果：", len(labels_num))
                print("测试平均准确率为：", avg_accuracy, "%")
```

数据集一：Phishing

结果：

```
第 1 次测试准确率为： 62.22222222222222 %
第 2 次测试准确率为： 61.48148148148148 %
第 3 次测试准确率为： 52.59259259259259 %
第 4 次测试准确率为： 56.52173913043478 %
第 5 次测试准确率为： 57.77777777777777 %
第 6 次测试准确率为： 53.48837209302325 %
第 7 次测试准确率为： 60.74074074074074 %
第 8 次测试准确率为： 52.59259259259259 %
第 9 次测试准确率为： 61.48148148148148 %
第 10 次测试准确率为： 48.148148148148145 %
样本数量为： 1218
测试数量为： 135
属性类为： 9
标签结果： 3
测试平均准确率为： 56.70471482604951 %
```

使用 sklearn：

```
第 1 次测试准确率为： 85.18518518518519 %
第 2 次测试准确率为： 77.77777777777779 %
第 3 次测试准确率为： 85.92592592592592 %
第 4 次测试准确率为： 78.51851851851852 %
第 5 次测试准确率为： 81.41592920353983 %
第 6 次测试准确率为： 78.51851851851852 %
第 7 次测试准确率为： 82.22222222222221 %
第 8 次测试准确率为： 81.48148148148148 %
第 9 次测试准确率为： 78.51851851851852 %
第 10 次测试准确率为： 85.18518518518519 %
样本数量为： 1218
测试数量为： 135
属性类为： 9
测试平均准确率为： 81.47492625368731 %
```

分析:

说明自己的模型不行，还需要改进，并且我没有使用极大似然法，模型太过简单，只是采用了先验概率来计算。

3.4.2、小结

关于朴素贝叶斯分类器，准确率有点低，在检查完代码后，觉得是朴素贝叶斯分类器的条件比较苛刻，sklearn 也只要 80%的准确率，要求各个特征独立，而且相等 ，还有就是和自己假设的模型也有关，所以我就值测试这一组值了。

4、总结

结合这些天对机器学习学习和理解，发现不仅仅是掌握算法那么简单，还需要学会优化其速度和准确度。自己的代码能力还有限，需要借助网上大佬们的代码。机器学习算法有很多，但最重要的是学会每类算法的核心思想。

另外，训练数据的处理和挑选也是一门学问，数据应该选取适合自己算法的。

模型的参数选择也很重要，比如 K-means 算法，直接影响到最终的结果。经验也很重要。

最后，实验过程中发现了一个有问题的数据集，也算是个小乐趣。有时候并不是全是自己的错。