数据挖掘技术报告

韩露露

一、聚类分析

实验一组算法，不少于 4 个，至少包含一个基于划分的、一个基于层次的、一个基于密度的，具体的算法自行选择。对给定的数据集进行聚类分析。

二、实验过程

1.环境要求

1.使用 python==3.6.12，python 的版本要满足 tensorflow 的 1.x 版本。
2.额外函数包 numpy==1.19.2，scikit-learn==0.23.2，matplotlib==3.3.2，pandas==1.1.5，tensorflow=1.12.0，jupyter=1.0.0，bert_serving==0.0.1，bert_serving_server==1.10.0。
3.系统版本 Windows10。

2.数据读取处理

2.1. 数据集 data.csv

2.1.1.捕获的数据集内容

引用的原始数据集由 5 个不同的文件夹组成，每个文件夹包含 100 个文件，每个文件代表一个人员。每个文件记录了大脑 23.6 秒内的数据。将相应的时间序列采样为 4097 个数据点，每个数据点是 EEG 记录在不同时间点的值。总计 500 个人，每人有 4097 个数据点，即 23.5 秒。将每 4097 个数据点划分并随机排列为 23 个区块，每个区块包含 178 个数据点，即 1 秒，每个数据点是 EEG 记录在不同时间点的值。因此，总计 23x500=11500 条信息（行），每个信息包含 1 秒（列）的 178 个数据点，最后一列 y 表示标签，即人的状态。
各个标签的内容如下：

5 - eyes open, means when they were recording the EEG signal of the brain the patient had their eyes open

4 - eyes closed, means when they were recording the EEG signal the patient had their eyes closed

3 - Yes they identify where the region of the tumor was in the brain and recording the EEG activity from the healthy brain area

2 - They recorder the EEG from the area where the tumor was located

1 - Recording of seizure activity

2.1.2.数据集分析

1.数据集的列数较大，坐标维度较大，可以考虑使用 PCA 降低维度。
2.数据的值差异比较大，可以进行预处理标准化。同时 PCA 前进行标准化很重要。PCA 通常是用于高维数据的降维，它可以将原来高维的数据投影到某个低维的空间上并使得其方差尽量大。如果数据其中某一特征（矩阵的某一列）的数值特别大，那么它在整个误差计算的比重上就很大，那么可以想象在投影到低维空间之后，为了使低秩分解逼近原数据，整个投影会去努力逼近最大的那一个特征，而忽略数值比较小的特征。因为在建模前我们并不知道每个特征的重要性，这很可能导致了大量的信息缺失。为了“公平”起见，防止过分捕捉某些数值大的特征，我们会对每个特征先进行标准化处理，使得它们的大小都在相同的范围内，然后再进行 PCA。此外，从计算的角度讲，PCA 前对数据标准化还有另外一个好处。因为 PCA 通常是数值近似分解，而非求特征值、奇异值得到解析解，所以当我们使用梯度下降等算法进行 PCA 的时候，我们最好先要对数据进行标准化，这是有利于梯度下降法的收敛。

3.数据的规模比较大，计算量复杂的算法可能没法有效的使用。

2.1.3.数据集读取和预处理

使用 pandas 读取 csv 文件并转成 numpy 数组，方便进行数据运算。使用 sklearn.decomposition.PCA 进行 PCA，使用 sklearn.preprocessing.StandardScaler 进行标准化。

```
# 返回数据和标签
def DataSet():
    # 使用pandas读取csv文件并转成numpy数组
    data = np.array(pd.read_csv('.\data\data.csv').values.tolist())
    # 提取X,Y数据
    X = data[:, 1:-1].astype(np.float32)
    Y = data[:, -1].astype(np.float32)
    # 标准化
    StdModel = StandardScaler()
    X = StdModel.fit_transform(X)
    # PCA
    PCAModel = PCA(n_components=34)
    X = PCAModel.fit_transform(X)
    return X, Y
```

处理完的数据为维度为 11500x34，所有数据标准化。

2.2. 数据集 issues.csv：

2.2.1. 数据集内容

数据的第一列为各种提示信息的文本，第二列为提示信息的发出者或者行为(系统，备份等)，第三列也是提示信息的发出者或者行为(系统，备份等)，不过扩大了范围。

2.2.2.数据集分析

对于文本的聚类，首先就要将句子转成维度相同的向量或者矩阵，进行操作。

2.2.3. 数据集读取和预处理

使用 pandas 读取 csv 文件并转成 numpy 数组，方便进行数据运算。使用 BERT 产生句子向量。

1.读取文本

由于文本中有很多的标点符号，我们聚类过程中不讨论标点带来的语义变化，所以处理文本删除所有的标点符号和首位空格，以减少大小和无用信息。

```
# 返回文本和标签
def TxtSet():
    data = np.array(pd.read_csv('.\data\issues.csv',
header=None).values.tolist())
    txt = data[:, 0].tolist()
    newtxt = []
    # 删除标点和前后空格
    for str in txt:
        for i in punctuation:
            str = str.replace(i, '')
        str.strip()
```

```
        newtxt.append(str)
    return newtxt
```

2.句子转成向量：

1.BERT 模型

BERT 的全称为 Bidirectional Encoder Representation from Transformers，是一个预训练的语言表征模型。它强调了不再像以往一样采用传统的单向语言模型或者把两个单向语言模型进行浅层拼接的方法进行预训练，而是采用新的 masked language model（MLM），以致能生成深度的双向语言表征。

该模型有以下主要优点：

1）采用 MLM 对双向的 Transformers 进行预训练，以生成深层的双向语言表征。

2）预训练后，只需要添加一个额外的输出层进行 fine-tune，就可以在各种各样的下游任务中取得 state-of-the-art 的表现。在这过程中并不需要对 BERT 进行任务特定的结构修改。

BERT 可以用于问答系统，情感分析，垃圾邮件过滤，命名实体识别，文档聚类等任务中，作为这些任务的基础设施即语言模型，BERT 的代码也已经开源：https://github.com/google-research/bert。可以对其进行微调，将它应用于我们的目标任务中，BERT 的微调训练也是快而且简单的。

2.搭建 BERT

安装完 bert-serving-server 和 bert-serving-client 后，从 https://github.com/google-research/bert#pre-trained-models 下载与训练模型，开启服务：

```
def bertServerStart():
    args = get_args_parser().parse_args(['-model_dir',
r'C:\Users\HanLulu\Desktop\CLUSTERING\uncased_L-12_H-768_A-12',
                                         '-port', '86500',
                                         '-port_out', '86501',
                                         '-max_seq_len', '512',
                                         '-mask_cls_sep',
                                         '-cpu'])
    bs = BertServer(args)
    bs.start()
```

由于文本的条数太多，为了减少训练时间和内存，从所有数据中随机选择 1000 条进行转换，为了方便多次调用，将转换的向量保存成 numpy 的数据文件

```
txt = TxtSet()
txt = random.sample(txt, 1000)
bc = BertClient(port=86500, port_out=86501,show_server_config=True)
vec = bc.encode(txt)
vec = np.array(vec)
np.save('word2vec.npy', vec)
```

同样进行标准化与 PCA

```
# 返回文本向量
def TxtVec():
    # 读取文本向量
```

```
    txt = np.load('word2vec.npy')
    # 标准化
    StdModel = StandardScaler()
    txt = StdModel.fit_transform(txt)
    # PCA
    PCAModel = PCA(n_components=14)
    txt = PCAModel.fit_transform(txt)
    print(np.nonzero(PCAModel.explained_variance_ratio_ >= 1e-2))
    return txt
```

处理后的数据维度是 1000x14。

2.3 建模验证数据

在进行算法理解和实现时，为了验证自己写的算法正确性，进行了数据模拟，测试数据为一组西瓜检测数据。

```
# 数据集：每三个是一组分别是西瓜的编号，密度，含糖量
data = '''
1,0.697,0.46,2,0.774,0.376,3,0.634,0.264,4,0.608,0.318,5,0.556,0.21
5,6,0.403,0.237,7,0.481,0.149,8,0.437,0.211,9,0.666,0.091,10,0.243,
0.267,11,0.245,0.057,12,0.343,0.099,13,0.639,0.161,14,0.657,0.198,1
5,0.36,0.37,16,0.593,0.042,17,0.719,0.103,18,0.359,0.188,19,0.339,0
.241,20,0.282,0.257,21,0.748,0.232,22,0.714,0.346,23,0.483,0.312,24
,0.478,0.437,25,0.525,0.369,26,0.751,0.489,27,0.532,0.472,28,0.473,
0.376,29,0.725,0.445,30,0.446,0.459
'''
a = data.split(',')
dataset = np.array([(float(a[i]),float(a[i+1])) for i in
range(1,len(a)-1,3)])
```

3. 模型建立与算法描述

常用的聚类方法

- **划分方法**首先创建 k 个分区的初始集合，其中参数 k 是要构建的分区数。然后，它采用迭代重定位技术，试图通过把对象从一个簇移到另一个簇来改进划分的质量。典型的划分方法包括 k - 均值、k - 中心点、CLARANS。
- **层次方法**创建给定数据对象集的层次分解。根据层次分解的形成方式，层次方法可以分为凝聚的（自底向上）或分裂的（自顶向下）。为了弥补合并或分裂的僵硬性，凝聚的层次方法的聚类质量可以通过以下方法改进：分析每个层次划分中的对象连接（如 Chameleon），或者首先执行微聚类（也就是把数据划分为“微簇”），然后使用其他的聚类技术，迭代重定位，在微簇上聚类（如 BIRCH）。
- **基于密度的方法**基于密度的概念来聚类对象。它或者根据邻域中对象的密度（例如 DBSCAN），或者根据某种密度函数（例如 DENCLUE）来生成簇。OPTICS 是一个基于密度的方法，它生成数据聚类结构的一个增广序。
- **基于网格的方法**首先将对象空间量化为有限数目的单元，形成网格结构，然后在网格结构上进行聚类。STING 是基于网格方法的一个典型例子，它基于存储在网格单元中的统计信息聚类。CLIQUE 是基于网格的子空间聚类算法。
- **聚类评估**估计在数据集上进行聚类分析的可行性和由聚类方法产生的结果的质量。任务包括评估聚类趋势、确定簇数和测定聚类的质量。

3.1.基于划分

3.1.1KMeans 法

Kmeans 算法又名 k 均值算法。其算法思想大致为：先从样本集中随机选取 k 个样本作为簇中心，并计算所有样本与这 k 个“簇中心”的距离，对于每一个样本，将其划分到与其距离最近的“簇中心”所在的簇中，对于新的簇计算各个簇的新的“簇中心”。

1.算法过程：

1)选取 K 个点做为初始聚集的簇心（也可选择非样本点）；
2)分别计算每个样本点到 K 个簇核心的距离（这里的距离一般取欧氏距离或余弦距离），找到离该点最近的簇核心，将它归属到对应的簇；
3)所有点都归属到簇之后， M 个点就分为了 K 个簇。之后重新计算每个簇的重心（平均距离中心），将其定为新的“簇核心”；
4)反复迭代 2)-3)步骤，直到达到某个中止条件。
注：常用的中止条件有迭代次数、最小平方误差 MSE、簇中心点变化率；

2.优缺点：

优点：

1)理解容易，聚类效果不错；
2)处理大数据集的时候，该算法可以保证较好的伸缩性和高效率；
3)当簇近似高斯分布的时候，效果非常不错。

缺点：

1)K 值是用户给定的，在进行数据处理前，K 值是未知的，给定合适的 k 值，
2)需要先验知识，凭空估计很困难，或者可能导致效果很差。
3)对初始簇中心点是敏感的。
4)不适合发现非凸形状的簇或者大小差别较大的簇。
5)特殊值(离群值或称为异常值)对模型的影响比较大。

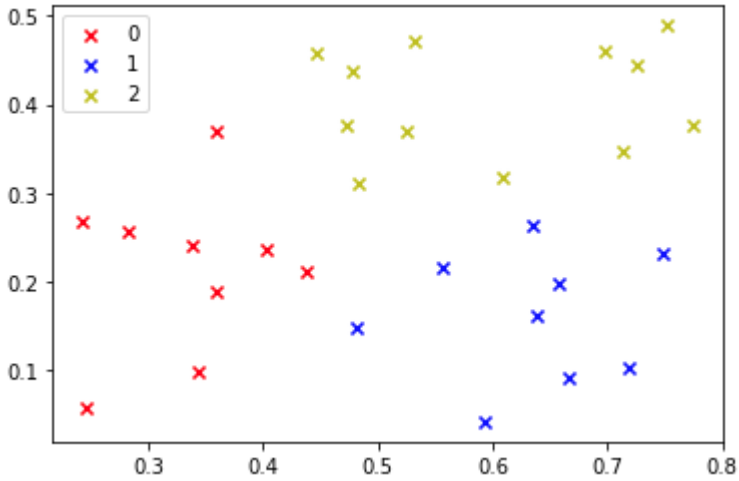
3.代码实现：

```
# 求两空间点的欧氏距离
def dist2(x, y):
    return np.sqrt(np.sum((x - y) ** 2))

# 随机质心
def randCenter(X, k):
    n = X.shape[1]  # 数据列数，即点坐标分量
    centroids = np.zeros((k, n))  # 创建中心点
    for j in range(n):  # 循环所有特征列，获得每个中心点该列的随机值
        valMin = np.min(X[:, j])  # 寻找列的最小值
        valRange = np.float(np.max(X[:, j]) - valMin)  # 寻找列值范围
        centroids[:, j] = valMin + valRange * np.random.rand(k, )  #
获得每列的随机值 一列一列生成
    return centroids

# K-Means方法
def kMeans(X, k):
    m = X.shape[0]  # 数据的个数
```

```
    n = X.shape[1]  # 数据的维度
    centroids = randCenter(X, k)  # 随机初始化质心点
    centIndxDist = np.zeros((m, 2))  # 距离矩阵：行数对应每个数据点数目，第一列是与该点距离最近的质心点索引，第二列是距离的平方
    changeFlag = True  # 聚类改变的标志
    while changeFlag:  # 不断循环直到稳定
        changeFlag = False
        for i in range(m):
            minDist = np.inf
            minIndex = -1
            for j in range(k):  # 遍历质心计算与数据点的距离
                dist = dist2(centroids[j, :], X[i, :])
                if dist < minDist:  # 找到与其距离最小的质心点，并在centIndxDist更新
                    minDist = dist
                    minIndex = j
            if centIndxDist[i, 0] != minIndex:  # 只要还在变化就继续更新
                changeFlag = True
            centIndxDist[i, :] = minIndex, minDist ** 2
        for ki in range(k):  # 更新质心
            pInCurrCluster = X[np.nonzero(centIndxDist[:, 0] == ki)]  # 找到所有距离该质心最近的点数据（每一个簇所拥有的所有数据集）
            if len(pInCurrCluster):  # 有数据则进行取平均值
                centroids[ki, :] = np.mean(pInCurrCluster, axis=0)
            else:  # 没有则置0
                centroids[ki, :] = np.zeros((n,))
    return centroids, centIndxDist
```

4. 结果测试

使用西瓜数据测试得到的效果良好：



3.1.2. 二分 KMeans 法

由于传统的 KMeans 算法的聚类结果容易受到初始聚类中心点选择的影响，因此在传统的 KMeans 算法的基础上进行算法改进，对初始中心点选取比较严格，各

中心点的距离较远，这就避免了初始聚类中心会选到一个类上，一定程度上克服了算法限入局部最优状态。

1.算法流程

1)使用 Kmeans(k=2)将数据集分成 2 个簇，记录 SSE

2)对于每一个簇来说，都有自己当前的 SSE，取名为父节点 SSE

a. 对这些簇都进行 Kmeans 二分类，并且记录分出的 2 个簇的 SSE 只和，称之为子节点总 SSE

b. 记录这个簇被 2 分类之后 SSE 的差值，SSE 差值 = 父节点 SSE - 子节点 SSE

3)选择 SSE 差值最大的那个簇进行划分，而其他的簇不进行划分。

4)重复第二的步骤，直到簇的总个数达到 K

2.优缺点

优点：

1)二分 K 均值算法可以加速 K-means 算法的执行速度，因为相似度计算少了

2)不受初始化问题的影响，因为随机点选取少了，且每一步保证误差最小

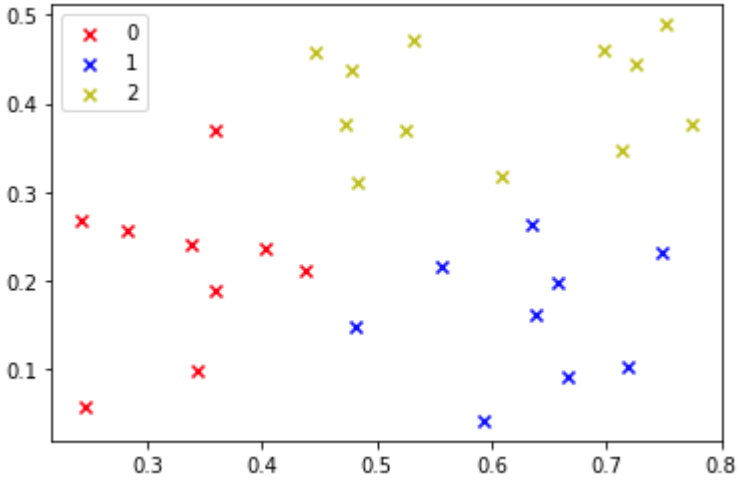
3.代码实现

```
# 二分k-Means方法
def biKMeans(X, k):
    m = X.shape[0]  # 数据的个数
    centIndxDist = np.zeros((m, 2))  # 距离矩阵：行数对应每个数据点数目，第一列是与该点距离最近的质心点索引，第二列是距离的平方
    centLists = []
    centLists.append(np.mean(X, axis=0))  # 初始质心为全体平均值
    for i in range(m):
        centIndxDist[i, 1] = dist2(centLists[0], X[i, :]) ** 2  # 计算每个点与质心点距离的平方，这里是最初的0簇
    while len(centLists) < k:
        lowestSSE = np.inf  #
        for ki in range(len(centLists)):
            pInCurrCluster = X[np.nonzero(centIndxDist[:, 0] == ki)]  # 找到所有距离该质心最近的点数据（每一个簇所拥有的所有数据集）
            centroidArray, splitIndxDist = kMeans(pInCurrCluster, 2)  # 进行内部进行2-means得到两个质心
            sseSplit = np.sum(splitIndxDist[:, 1])  # 当前簇划分的SSE
            sseNotSplit = np.sum(centIndxDist[np.nonzero(centIndxDist[:, 0] != ki), 1])  # 不再当前簇划分的SSE
            if (sseSplit + sseNotSplit) < lowestSSE:  # 两种方差和小于先前的SSE，说明这种分配方式减小了误差率，可以更新
                bestCentSplit = ki  # 当前簇划分设为最佳
                bestCents = centroidArray  # 当前的划分质心设为最好
                bestIndxDist = splitIndxDist.copy()
                lowestSSE = sseSplit + sseNotSplit  # 更新小的SSE
        bestIndxDist[np.nonzero(bestIndxDist[:, 0] == 1), 0] =
```

```
len(centLists)  # 2-means返回系数0或1,需要把1换成多出的那个簇数目
        bestIndxDist[np.nonzero(bestIndxDist[:, 0] == 0), 0] =
bestCentSplit  # 把0换成最好的那个簇数
        centLists[bestCentSplit] = bestCents[0, :]  # 将最好的那个簇数的
质心更新
        centLists.append(bestCents[1, :])  # 添加多出的质心
        centIndxDist[np.nonzero(centIndxDist[:, 0] ==
bestCentSplit), :] = bestIndxDist  # 更新距离矩阵
    return centLists, centIndxDist
```

4.结果测试

使用西瓜数据测试得到的效果良好：



3.2.基于层次：

3.2.1.AGNES 算法

AGNES(Agglomerative Nesting) 是凝聚的层次聚类算法，如果簇 C1 中的一个对象和簇 C2 中的一个对象之间的距离是所有属于不同簇的对象间欧式距离中最小的，C1 和 C2 可能被合并。这是一种单连接方法，其每个簇可以被簇中的所有对象代表，两个簇之间的相似度由这两个簇中距离最近的数据点对的相似度来确定。

1.算法流程

1)将每个对象初始化为簇对象

2)将最近的两个簇合并，使簇的总数量减一

3)重复第二步直至达到理想簇数量

2.优缺点

缺点：

1)算法复杂度为 $O(N^2)$，对于大规模数据不适用。

2)当多个簇距离相同时，簇合并选择困难。

3)算法无法回溯，不具很好的可收缩性。

3.代码实现

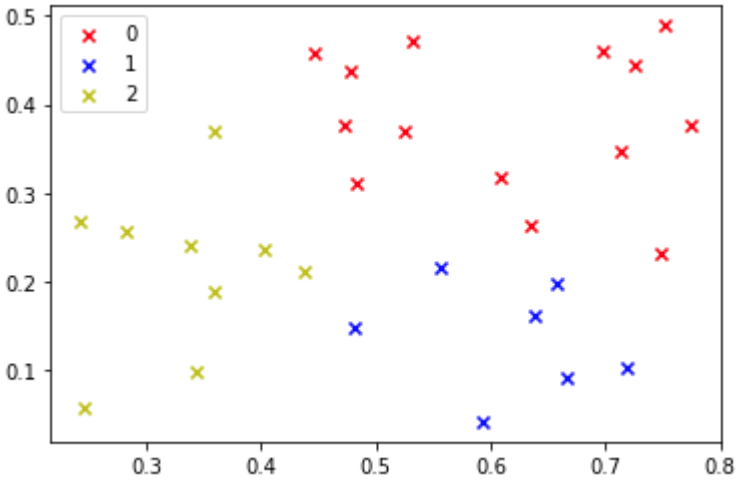
```
# 求两空间点的欧氏距离
def dist2(x, y):
    return np.sqrt(np.sum((x - y) ** 2))
```

```
# 求出最小距离
def minDist(Ci, Cj):
    return np.min([dist2(i, j) for i in Ci for j in Cj])

# AGNES算法
def AGNES(X, k, distmode='max'):
    if distmode == 'max':
        dist = maxDist
    elif distmode == 'min':
        dist = minDist
    elif distmode == 'avg':
        dist = avgDist
    # 初始化C
    C = []
    # 开始C就是各个点的簇
    for p in X:
        Ci = []
        Ci.append(p.tolist())
        C.append(Ci)
    # 计算所有簇间的最大距离
    m = len(C)
    # 距离矩阵为m*m,上三角
    M = np.zeros((m, m))
    for i in range(len(C) - 1):
        for j in range(i + 1, len(C)):
            M[i][j] = dist(np.array(C[i]), np.array(C[j]))
    # 转置上三角后进行合并矩阵得到M，减少计算量
    q = len(X)
    # 合并更新
    while q > k:
        # 找最近的两个簇
        x, y, min = findMinIndex(M)
        # 合并簇
        C[x].extend(C[y])
        # 删除被合并的,序号自动更改
        C.remove(C[y])
        # 删除距离矩阵的y行，y列
        M = np.delete(M, y, axis=1)
        M = np.delete(M, y, axis=0)
        # 计算合并新的簇，与其他的距离
        # 在x行之前都是行间计算
        for i in range(x):
            M[i][x] = dist(np.array(C[i]), np.array(C[x]))
```

```
        for j in range(len(M[x])):
            M[x][j] = dist(np.array(C[j]), np.array(C[x]))
        # 每次簇个数减少1
        q -= 1
    return C
```

4.结果测试

使用西瓜数据测试得到的效果一般，右下脚蓝色的分类出现问题，主要原理就是合并过程中出现选择困难，合并错误后不可逆，将一直错下去。



3.3.基于密度：

3.3.1.DBSCAN 算法

DBSCAN 是一种基于密度的空间聚类算法。该算法将具有足够密度的区域划分为簇，并在具有噪声的空间数据库中发现任意形状的簇，它将簇定义为密度相连的点的最大集合。该算法利用基于密度的聚类的概念，即要求聚类空间中的一定区域内所包含对象（点或其他空间对象）的数目不小于某一给定阈值。

1.算法流程：

1)将所有点标记为核心点、边界点或噪声点；
2)删除噪声点；
3)为距离在 Eps 之内的所有核心点之间赋予一条边；
4)每组连通的核心点形成一个簇；
5)将每个边界点指派到一个与之关联的核心点的簇中（哪一个核心点的半径范围之内）

2.优缺点

优点：

1)聚类速度快且能够有效处理噪声点和发现任意形状的空间聚类；
2)与 K-MEANS 比较起来，不需要输入要划分的聚类个数；
3)聚类簇的形状没有偏倚；
4)可以在需要时输入过滤噪声的参数。

缺点：

1)当数据量增大时，要求较大的内存支持 I/O 消耗也很大；
2)当空间聚类的密度不均匀、聚类间距差相差很大时，聚类质量较差，因为这种情况下参数 MinPts 和 Eps 选取困难

3)算法聚类效果依赖与距离公式选取，实际应用中常用欧式距离，对于高维数据，存在“维数灾难”。

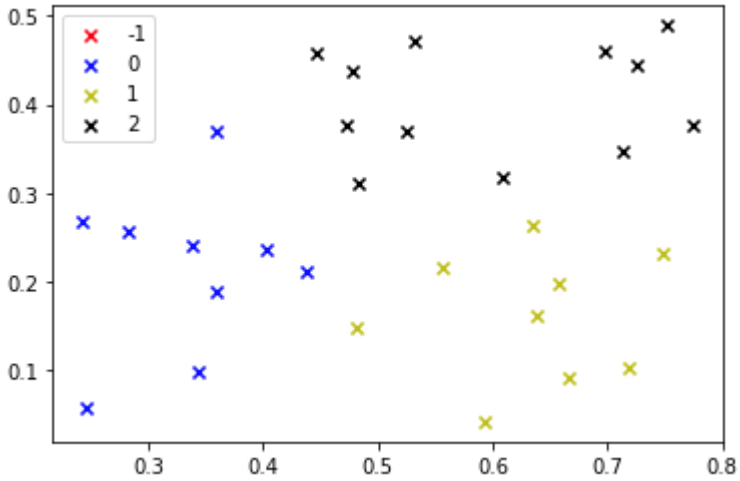
3.代码实现

```
# vistList类用于记录访问列表
class vistList:
    def __init__(self, count=0):
        self.unvisitedList = [i for i in range(count)]
        self.visitedList = []
        self.unvisitedNum = count

    def visit(self, pointId):
        self.visitedList.append(pointId)
        self.unvisitedList.remove(pointId)
        self.unvisitedNum -= 1


def DBSCAN(X, eps, minPts):
    nPoints = X.shape[0]
    # 标记所有对象为unvisited
    vPoints = vistList(count=nPoints)
    # 初始化簇标记列表C，簇标记为 k
    k = -1
    C = [-1 for i in range(nPoints)]
    # 构建KD-Tree，并生成所有距离<=eps的点集合
    kd = KDTree(X)
    while vPoints.unvisitedNum > 0:
        # 随机选择一个unvisited对象p
        p = random.choice(vPoints.unvisitedList)
        # 标t己p为visited
        vPoints.visit(p)
        # N是p的epsilon-邻域点列表
        N = kd.query_ball_point(X[p], eps)
        # 如果p的epsilon邻域至少有MinPts个对象
        if len(N) >= minPts:
            # 创建个一个新簇C，并把p添加到C
            # 对标记列表第p个结点进行赋值
            k += 1
            C[p] = k
            for p1 in N:
                # p1是unvisited
                if p1 in vPoints.unvisitedList:
                    # 标记p1为visited
                    vPoints.visit(p1)
                    # M是p1的epsilon-邻域
```

```
                    M = kd.query_ball_point(X[p1], eps)
                    # 如果p1的epsilon-邻域至少有MinPts个点，把这些点去重添加
到N
                    if len(M) >= minPts:
                        for i in M:
                            if i not in N:
                                N.append(i)
                    # 如果p1还不是任何簇的成员，把p1添加到c
                    if C[p1] == -1:
                        C[p1] = k
        # 否则标记p为噪声
        else:
            C[p] = -1
    return C
```

4.结果测试

使用西瓜数据测试如下质量良好，其中-1 标签都是噪声点，没有聚类，从图中可以看出，DBSCAN 算法对参数的选择很重要，两参数的选取是聚类质量的关键。



4.大规模聚类与评价

对于大规模数据，手写代码显然不合适，所以对于所给数据集，本项目使用第三方库 sklearn。Scikit-learn(sklearn)是机器学习中常用的第三方模块，对常用的机器学习方法进行了封装，包括回归(Regression)、降维(Dimensionality Reduction)、分类(Classfication)、聚类(Clustering)等方法。

4.1. sklearn 聚类函数

对前述的模型调用相应的 sklearn 算法，其中对 KMeans 的改进使用的是 MiniBatchKMeans，而不是二分 KMeans 方法。由于 DBSCAN 的调参较为困难，替换为使用改进的 OPTICS 算法。

4.1.1.KMeans 算法函数

函数调用：

sklearn.cluster.KMeans

主要参数：
n_clusters:质心数量，也就是分类数。

4.1.2.MiniBatchKMeans 算法函数

MiniBatchKMeans 聚类与传统 KMeans 的区别是，原来对质心的更新是单点进行，现在改为对一小批数据进行更新及计算质心。（速度比 K-means 快，精度更低）
函数调用：
sklearn.cluster.MiniBatchKMeans
主要参数：
n_clusters:质心数量，也就是分类数。

4.1.3 AGENS 算法函数

函数调用：
sklearn.cluster. AgglomerativeClustering
主要参数：
n_clusters：分类数。

4.1.4. OPTICS 算法函数

OPTICS（基于点排序来识别聚类结构）算法与 DBSCAN 算法有许多相似之处，可视为 DBSCAN 的泛化，将 eps 要求从单个值放宽到一个范围。DBSCAN 和 OPTICS 之间的主要区别在于，OPTICS 算法构建了一个可 reachability graph，为每个样本分配一个 reachability_距离，和在群集 ordering_属性中的一个点; 这两个属性在拟合(fit)模型时分配，并用于确定 cluster 的成员。
函数调用：
sklearn.cluster.OPTICS
主要参数（详细参数）：
eps:两个样本之间的最大距离，即扫描半径
min_samples ：作为核心点的话邻域(即以其为圆心，eps 为半径的圆，含圆上的点)中的最小样本数(包括点本身)。

4.2.sklearn 聚类评价函数

4.2.1 轮廓系数（Silhouette Coefficient）

轮廓系数（Silhouette Coefficient），是聚类效果好坏的一种评价方式。最早由 Peter J. Rousseeuw 在 1986 提出。它结合内聚度和分离度两种因素。可以用来在相同原始数据的基础上用来评价不同算法、或者算法不同运行方式对聚类结果所产生的影响。

1.具体方法：

1）计算样本 i 到同簇其他样本的平均距离 ai。ai 越小，说明样本 i 越应该被聚类到该簇。将 ai 称为样本 i 的簇内不相似度。某一个簇 C 中所有样本的 a i 均值称为簇 C 的簇不相似度。
2）计算样本 i 到其他某簇 Cj 的所有样本的平均距离 bij，称为样本 i 与簇 Cj 的不相似度。定义为样本 i 的簇间不相似度：bi =min{bi1, bi2, ..., bik}，即某一个样本的簇间不相似度为该样本到所有其他簇的所有样本的平均距离中最小的那一个。
bi 越大，说明样本 i 越不属于其他簇。
3）根据样本 i 的簇内不相似度 a i 和簇间不相似度 b i ，定义某一个样本样本 i 的轮廓系数：

$$s(i)=\frac{b(i)-a(i)}{\max\{a(i),b(i)\}} \qquad s(i)=\begin{cases}1-\dfrac{a(i)}{b(i)}, & a(i)<b(i)\\ 0, & a(i)=b(i)\\ \dfrac{b(i)}{a(i)}-1, & a(i)>b(i)\end{cases}$$

4）判断：
si 接近 1，则说明样本 i 聚类合理；
si 接近-1，则说明样本 i 更应该分类到另外的簇；
若 si 近似为 0，则说明样本 i 在两个簇的边界上。
5） 所有样本的轮廓系数 S
所有样本的 s i 的均值称为聚类结果的轮廓系数，定义为 S，是该聚类是否合理、有效的度量。聚类结果的轮廓系数的取值在[-1,1]之间，值越大，说明同类样本相距约近，不同样本相距越远，则聚类效果越好。

2.优缺点：

优点：

1)对于不正确的 clustering （聚类），分数为 -1 ， highly dense clustering （高密度聚类）为 +1 。零点附近的分数表示 overlapping clusters （重叠的聚类）。
2)当 clusters （簇）密集且分离较好时，分数更高，这与 cluster （簇）的标准概念有关。

缺点

1)convex clusters（凸的簇）的 Silhouette Coefficient 通常比其他类型的 cluster （簇）更高，例如通过 DBSCAN 获得的基于密度的 cluster（簇）。

3.sklearn 函数

函数调用：
sklearn.metrics.silhouette_score
主要参数：
X：表示要聚类的样本数据，一般形如（samples，features）的格式
labels：即聚类之后得到的 label 标签，形如（samples,）的格式
metric：默认是欧氏距离

4.2.2.CH 分数（Calinski Harabasz Score ）

计算简单直接，得到的 Calinski-Harabasz 分数值 ss 越大则聚类效果越好。

1.具体算法：

1)Calinski-Harabasz 分数值 ss 的数学计算公式是：

$$s(k)=\frac{\mathrm{Tr}(B_k)}{\mathrm{Tr}(W_k)}\times\frac{N-k}{k-1}$$

其中 Bk 称之为 between-clusters dispersion mean （簇间色散平均值）
Wk 称之为 within-cluster dispersion（群内色散之间）
2)它们的计算公式如下：

$$W_k = \sum_{q=1}^{k} \sum_{x \in C_q} (x - c_q)(x - c_q)^T$$

$$B_k = \sum_{q} n_q (c_q - c)(c_q - c)^T$$

类别内部数据的协方差越小越好，类别之间的协方差越大越好，这样的 Calinski-Harabasz 分数会高。 总结起来一句话：CH index 的数值越大越好。在真实的分群 label 不知道的情况下，可以作为评估模型的一个指标。 同时，数值越小可以理解为：组间协方差很小，组与组之间界限不明显。与轮廓系数的对比，最大的优势就是速度很快。

2.优缺点

优点:

1)当 cluster （簇）密集且分离较好时，分数更高，这与一个标准的 cluster （簇）有关。

2)得分计算很快

缺点:

2)凸的簇的 Calinski-Harabaz index（Calinski-Harabaz 指数）通常高于其他类型的 cluster（簇），例如通过 DBSCAN 获得的基于密度的 cluster （簇）。

3.sklearn 函数

函数调用：

sklearn.metrics.calinski_harabaz_score

主要参数：

X：表示要聚类的样本数据，一般形如（samples，features）的格式

labels：即聚类之后得到的 label 标签，形如（samples,）的格式

4.2.3 戴维森堡丁指数(DBI)

戴维森堡丁指数(DBI)，又称为分类适确性指标，是由大卫 L·戴维斯和唐纳德·Bouldin 提出的一种评估聚类算法优劣的指标。首先假设我们有 m 个时间序列，这些时间序列聚类为 n 个簇。m 个时间序列设为输入矩阵 X，n 个簇类设为 N 作为参数传入算法。使用下列公式进行计算：

$$DBI = \frac{1}{N} \sum_{i=1}^{N} \max_{j \neq i} \left(\frac{\overline{S_i} + \overline{S_j}}{\|w_i - w_j\|_2} \right)$$

这个公式的含义是度量每个簇类最大相似度的均值。

1.具体算法：

1）计算 Si。DBI 定义了一个分散度的值 Si：表示第 i 个类中，度量数据点的分散程度，DBI 计算公式中首先定义了 Si 变量，Si 计算的是类内数据到簇质心的平均距离，代表了簇类 i 中各样本的分散程度，计算公式为：

$$S_i = \left(\frac{1}{T_i} \sum_{j=1}^{T_i} |X_j - A_i|^p \right)^{1/p}$$

其中 Xj 代表簇类 i 中第 j 个数据点，也就是一个样本点，Ai 是簇类 i 的质心，Ti 是簇类 i 中数据的个数，p 在通常情况下取 2，这样就可以计算独立的数据点和质心的欧式距离（euclidean metric），此时表示各点到中心距离的标

准差，它们都可以用来衡量分散程度当然在考察流型和高维数据的时候，欧氏距离也许不是最佳的距离计算方式，但也是比较典型的了。
2）计算 Mij。DBI 定义了一个距离值 Mij：表示第 i 类与第 j 类的距离，分子之和计算完后，需计算分母 Mij，定义为簇类 i 与簇类 j 的距离，计算公式为：

$$M_{i,j} = ||A_i - A_j||_p = \left(\sum_{k=1}^{n} |a_{k,i} - a_{k,j}|^p\right)^{\frac{1}{p}}$$

3）计算 Rij。DBI 定义了一个相似度的值 Rij：Rij 衡量第 i 类与第 j 类的相似度。计算公式为：

$$R_{ij} \equiv \frac{S_i + S_j}{M_{ij}}$$

4）计算 DBI。通过以上公式的计算，我们再从 Rij 中选出最大值 Ri=max（Rij），即，第 i 类与其他类的相似度中最大的相似度的值。最后计算每个类的这些最大相似度的均值，便得到了 DBI 指数：

$$\bar{R} \equiv \frac{1}{N}\sum_{i=1}^{N} R_i$$

2. sklearn 函数
函数调用：
sklearn.metrics.davies_bouldin_score
主要参数：
X：表示要聚类的样本数据，一般形如（samples，features）的格式
labels：即聚类之后得到的 label 标签，形如（samples,）的格式
4.3. 测试数据集：
4.3.1 代码如下：

```
# 使用KMeans方法
def useKMeans(X, k):
    model = cluster.KMeans(n_clusters=k)
    yhat = model.fit_predict(X)
    return yhat


# 使用MiniBatchKMeans
def useMiniBatchKMeans(X, k):
    model = cluster.MiniBatchKMeans(n_clusters=k)
    yhat = model.fit_predict(X)
    return yhat


# 使用AGENS
def useAGENS(X, k):
    model = cluster.AgglomerativeClustering(n_clusters=k)
```

```
    yhat = model.fit_predict(X)
    return yhat


# 使用OPTICS
def useOPTICS(X,min_samples):
    model = cluster.OPTICS(min_samples=min_samples)
    yhat = model.fit_predict(X)
    return yhat


# 评估函数
def estimate(X, c):
    s1 = metric.silhouette_score(X, c, metric='euclidean')  # 计算轮
廓系数
    s2 = metric.calinski_harabasz_score(X, c)  # 计算CH score
    s3 = metric.davies_bouldin_score(X, c)  # 计算 DBI
    return s1, s2, s3


# 运行所有
def runAll(X,k,min_samples=5):
    # 使用KMeans
    print('使用KMeans:')
    start = time.perf_counter()
    c1 = useKMeans(X, k)
    end = time.perf_counter()
    ret = estimate(X, c1)
    print('聚类标签:', np.unique(c1))
    print("轮廓系数:{},CH:{},DBI:{}".format(ret[0], ret[1], ret[2]))
    print('Running time: %s Seconds' % (end - start))

    # 使用MiniBatchKMeans
    print('使用MiniBatchKMeans:')
    start = time.perf_counter()
    c2 = useMiniBatchKMeans(X, k)
    end = time.perf_counter()
    ret = estimate(X, c2)
    print('聚类标签:', np.unique(c2))
    print("轮廓系数:{},CH:{},DBI:{}".format(ret[0], ret[1], ret[2]))
    print('Running time: %s Seconds' % (end - start))

    # 使用AGENS
    print('使用AGENS:')
```

```
    start = time.perf_counter()
    c3 = useAGENS(X, k)
    end = time.perf_counter()
    ret = estimate(X, c3)
    print('聚类标签:', np.unique(c3))
    print("轮廓系数:{},CH:{},DBI:{}".format(ret[0], ret[1], ret[2]))
    print('Running time: %s Seconds' % (end - start))

    # 使用OPTICS
    print('使用OPTICS:')
    start = time.perf_counter()
    c4 = useOPTICS(X,min_samples)
    end = time.perf_counter()
    ret = estimate(X, c4)
    print('聚类标签:', np.unique(c4))
    print("轮廓系数:{},CH:{},DBI:{}".format(ret[0], ret[1], ret[2]))
    print('Running time: %s Seconds' % (end - start))

if __name__ == '__main__':
    # 读取数据
    X, Y = DataSet()
    txt = TxtSet()
    # 运行数据集1
    runAll(X,5)
    # 运行数据集2
    runAll(txt,8,13)
```

4.3.2. 运行结果：

1. 数据集 1 运行结果：

```
D:\Environment\Anaconda3\envs\bert\pythonw.exe C:/Users/HanLulu/Desktop/CLUSTERING/clustering.py
使用KMeans:
聚类标签: [0 1 2 3 4]
轮廓系数:0.0836697593331337,CH:270.75565139007375,DBI:3.145910382195846
Running time: 0.6382531 Seconds
使用MiniBatchKMeans:
聚类标签: [0 1 2 3 4]
轮廓系数:0.0657116025686264,CH:155.24174620155313,DBI:3.449551698812338
Running time: 0.1701596000000003 Seconds
使用AGENS:
聚类标签: [0 1 2 3 4]
轮廓系数:0.06413779407739639,CH:153.27976859016414,DBI:3.2764993022207802
Running time: 6.5560108 Seconds
使用OPTICS:
聚类标签: [-1  0  1]
轮廓系数:-0.41257843375205994,CH:0.5924980154932887,DBI:2.954973374743597
Running time: 99.57597439999999 Seconds
```

2. 数据集 2 运行结果：

```
使用KMeans:
聚类标签: [0 1 2 3 4 5 6 7]
轮廓系数:0.3551262319087982,CH:294.6327998603036,DBI:1.333414749485169
Running time: 0.09457209999999999 Seconds
使用MiniBatchKMeans:
聚类标签: [0 1 2 3 4 5 6 7]
轮廓系数:0.31465595960617065,CH:284.4899797184805,DBI:1.621380010967595
Running time: 0.024706400000000003 Seconds
使用AGENS:
聚类标签: [0 1 2 3 4 5 6 7]
轮廓系数:0.30454689264297485,CH:272.35285461225834,DBI:1.6875060633384988
Running time: 0.03447140000000001 Seconds
使用OPTICS:
D:\Environment\Anaconda3\envs\bert\lib\site-packages\sklearn\cluster\_optics.py:804:
  ratio = reachability_plot[:-1] / reachability_plot[1:]
聚类标签: [-1  0  1  2  3  4  5  6]
轮廓系数:0.0581674762070179,CH:109.45119262088886,DBI:1.0678987262094344
Running time: 0.8178892 Seconds
```

从运行速度上看 MiniBatchKMeans 有绝对优势，从聚类效果指标来看 KMeans 的得分最好。当数据集较大，类别较多时，AGNES 的速度也较快。OPTICS 在上述数据中的效果较差，速度和质量上都不怎么好，其主要原因在于两个超参数的调参较为困难，使用默认的参数效果很差。

三、总结与分析

1.基于划分

基于划分的方法：其原理简单来说就是，想象你有一堆散点需要聚类，想要的聚类效果就是“类内的点都足够近，类间的点都足够远”。首先你要确定这堆散点最后聚成几类，然后挑选几个点作为初始中心点，再然后依据预先定好的启发式算法（heuristic algorithms）给数据点做迭代重置（iterative relocation），直到最后到达“类内的点都足够近，类间的点都足够远”的目标效果。也正是根据所谓的“启发式算法”，形成了 k-means 算法及其变体包括 k-medoids、k-modes、k-medians、kernel k-means 等算法。

基于划分方法对于大型数据集是简单高效的、其时间复杂度、空间复杂度低。但是需要预先设定 K 值，对 K 值敏感，对噪声和离群值非常敏感，同时没办法处理不规则的聚类。

2.基于密度

基于密度的方法：k-means 解决不了不规则形状的聚类。于是就有了 Density-based methods 来系统解决这个问题。该方法同时也对噪声数据的处理比较好。其原理简单说画圈儿，其中要定义两个参数，一个是圈儿的最大半径，一个是一个圈儿里最少应容纳几个点。只要邻近区域的密度（对象或数据点的数目）超过某个阈值，就继续聚类,最后在一个圈里的，就是一个类。DBSCAN（Density-Based Spatial Clustering of Applications with Noise）就是其中的典型。

基于密度的方法对噪声不敏感；能发现任意形状的聚类。但是聚类的结果与参数有很大的关系，调参很麻烦。

3.基于层次

层次聚类主要有两种类型：合并的层次聚类和分裂的层次聚类。前者是一种自底向上的层次聚类算法，从最底层开始，每一次通过合并最相似的聚类来形成

上一层次中的聚类，整个当全部数据点都合并到一个聚类的时候停止或者达到某个终止条件而结束，大部分层次聚类都是采用这种方法处理。后者是采用自顶向下的方法，从一个包含全部数据点的聚类开始，然后把根节点分裂为一些子聚类，每个子聚类再递归地继续往下分裂，直到出现只包含一个数据点的单节点聚类出现，即每个聚类中仅包含一个数据点。
基于层次的方法缺点明显，时间复杂度高，$o(m^3)$，m 为点的个数；贪心算法的缺点，一步错步步错，但是可解释性好。