Data Mining 实验报告

- 1. VSM and KNN
  - 运行代码
  - 实验环境与实验数据
  - 实验目的
  - 实验步骤
    - 数据划分
    - 词干提取与词频统计
    - 建立向量空间
    - KNN - 验证集调整参数
    - 测试数据
  - 耗时问题
  - 代码实现
  - 实验结果
    - 暴力搜索
    - 用BallTree 对比
    - 最终结果
- 2. Naive Bayesian Classification
  - 运行代码
  - 实验环境与实验数据
  - 实验目的
  - 实验步骤
    - 数据划分
    - 词干提取与词频统计
    - 模型训练
    - 测试模型
  - 遇到的问题
  - 实验结果
    - 多项式模型结果
    - 伯努利模型结果
    - 混合模型结果
- 3. sklearn 聚类算法
  - 运行代码
  - 实验环境与实验数据
  - 实验目的
  - 实验步骤
    - 读取数据
    - 聚类及评估
      1. KMeans
      2. AffinityPropagation
      3. MeanShift
      4. SpectralClustering
      5. AgglomerativeClustering
      6. DBSCAN
      7. GaussianMixture
  - 实验结果

1. VSM and KNN

项目地址

运行代码

```
python src/knn.py
```

实验环境与实验数据

- CPU -- i5-4570
- 内存 -- 8GB
- 实验数据: 20news-18828

实验目的

通过对已有标签的文档进行操作，最终实现对无标签的文档进行正确分类。

实验步骤

数据划分

- 60% 的 Train 数据
- 20% 的 Cross-Validation 数据
- 20% 的 Test 数据

为保证数据的均匀分布，从每个类中挑选相同比例的数据组成训练集、验证集、测试集。

实现文件：『src/SplitData.py』

词干提取与词频统计

首先需要对实验中所有文档进行词干提取和词频统计，这里是将数据包括训练集、验证集、测试集全部进行统计，由于实验数据量过大，提取词干和统计词频所耗费的时间会很多，为了节省时间，将所有统计结果保存到文件中，以便用之后的每次实验。

运用『TextBlob』进行词干提取得到一个list，再用『collections.Counter』对list的每项进行计数，存储到一个字典中，键值为 文件名，对应的值为词频结果。统计结果保存到『data/tf_15_5.txt』。

实现文件：『src/utils.py』

建立向量空间

统计所有 Train 文档中出现的词干，由于文档数目太多，词干的数量会很大，所以对词干进行词频过滤，过滤规则有两条：

1. 需要在5个以上的文档中出现
2. 需要出现15次以上

过滤完成之后，就建立了一个维数与词干数目相同的空间，每个文档在这个空间中由一个向量表示，这就是『Vector Space』。并且这样一来，完成了训练操作，每个训练数据都对应一个向量。

实现文件：『src/vsm.py』

KNN - 验证集调整参数

读取验证集数据，像上一节一样，得到对应的向量，通过比较差异，计算向量之间的距离，再投票得到每条数据应该属于哪一类文档中。这里计算差异采用了两种方法：

1. 欧氏距离
2. 余弦相似度

经过验证发现，欧氏距离的分类的正确率没有余弦相似度计算分类的正确率高。而 K 值的变化对于验证集分类正确率没有太大的影响，分类的正确率会上下浮动 1-2%。选择一个表现最好的K值作为下一步测试数据所用的 K 值。

实现文件：『src/knn.py』

测试数据

测试数据步骤与在验证集上验证的步骤相同，最终将在测试数据上的分类精度记录下来。

耗时问题

耗时主要在两部分：

- 统计词干和词频，主要原因是因为IO操作太过耗时，并且文档数目过多。
- KNN，查找距离测试点最近几个点的标签，主要原因是 暴力搜索 的时间复杂度太高。

解决方法如下：

- 对于统计词干和词频遇到的耗时问题，采用『一劳永逸』的方法，将全部文档的数据一次统计完成，保存到文件中。
- 对于 KNN 耗时问题，采用KD-Tree，减少KNN 耗时，但KD-Tree在高维的表现并不是很好，所以采用Ball-Tree。

当使用BallTree的时候，由于余弦相似度在KNN上的表现要比欧式距离好，所以想采用余弦相似度来作为创建BallTree的metric，但是由于sklearn中没有余弦相似度。通过计算发现:

$cosine(d_i, d_j) = \frac{v_{d_i}^T v_{d_j}}{|v_{d_i}||v_{d_j}|}$

$dist(d_i, d_j) = \sqrt{\sum (v_{d_i} - v_{d_j})^2}$

当对向量进行归一化之后，$|v_{d_i}| = 1$、$|v_{d_j}| = 1$

而$dist^2(d_i, d_j) = \sum (v_{d_i} - v_{d_j})^2 = (v_{d_i} - v_{d_j}) * (v_{d_i} - v_{d_j})^T = 2 - 2v_{d_i}^T v_{d_j}$

与$cosine(d_i, d_j) = v_{d_i}^T v_{d_i}$的效果相同的，所以先将向量归一化，再计算欧氏距离就可以将余弦相似度用于BallTree中。

代码实现

代码结构如下：

|── data

| ├── tf_15_5.txt

| ├── train ……

| ├── cv ……

```
│ └── test ......
|── log
│ ├── logout-1.log
│ ├── logout-2.log
│ └── logout-3.log
├── src
│ ├── SplitData.py
│ ├── __pycache__
│ │ ├── utils.cpython-35.pyc
│ │ ├── utils.cpython-36.pyc
│ │ ├── vsm.cpython-35.pyc
│ │ └── vsm.cpython-36.pyc
│ ├── knn.py
│ ├── test.py
│ ├── utils.py
│ ├── utils.pyc
└── vsm.py
```

其中『data』文件夹存放数据，包括：

- 所有文件的词干和词频 『tf_15_5.txt』
- 训练数据『train』
- 验证数据『cv』
- 测试数据『test』

『log』文件夹存放程序运行的日志文件：

- 『logout-1.log』是没有采用BallTree的方法，暴力搜索，耗时 47622s。
- 『logout-2.log』是采用了BallTree的方法，跑完一遍验证集的日志。
- 『logout-3.log』是将程序完成之后运行的日志。

『src』文件夹存放源码：

- SplitData.py 的作用是划分数据集
- utils.py 提供了读取文件、获取一个目录下所有文件词干和词频、建立词典确立维数的函数。
- vsm.py 提供了建立向量空间的函数。
- knn.py 是程序的主体，里面实现了实验流程。
- test.py 是测试程序子功能的中间文件。

实验结果

暴力搜索

> predict 3765 data has cost time 47622.646603s Predict Accuracy is 82.416999%

用**BallTree** 对比

> predict 3765 data has cost time 418.819660s Predict Accuracy is 84.780876%

最终结果

> Start Choose K value
>
> Start evaluate k value is 2
>
> k value is 2,Predict Accuracy is 82.549801%
>
> Start evaluate k value is 3
>
> k value is 3,Predict Accuracy is 83.771580%
>
> Start evaluate k value is 4
>
> k value is 4,Predict Accuracy is 84.037185%
>
> Start evaluate k value is 5
>
> k value is 5,Predict Accuracy is 84.515272%
>
> Start evaluate k value is 6
>
> k value is 6,Predict Accuracy is 84.249668%
>
> Start evaluate k value is 7
>
> k value is 7,Predict Accuracy is 84.488712%
>
> Start evaluate k value is 8
>
> k value is 8,Predict Accuracy is 84.249668%
>
> Start evaluate k value is 9 k value is 9,Predict Accuracy is 84.249668%
>
> Choose K value has cost time 3273.398342s
>
> max accuracy is 84.515272%, k value is 5
>
> Start Predict Predict has cost time 408.537486s Predict Accuracy is 84.701493%

2. Naive Bayesian Classification

项目地址

运行代码

```
python src/nbc.py
```

实验环境与实验数据

- CPU -- i5-4570
- 内存 -- 8GB
- 实验数据: 20news-18828

实验目的

运用 Naive Bayesian Classification 的方法通过对已有标签的文档进行操作，最终实现对无标签的文档进行正确分类。

实验步骤

数据划分

- 60% 的 Train 数据
- 40% 的 Test 数据

为保证数据的均匀分布，从每个类中挑选相同比例的数据组成训练集、测试集。沿用了上一个 Homework 采用的数据集，由于 NBC 方法中所调超参数较少，所以将验证集与测试集合并为测试集。

词干提取与词频统计

首先需要对实验中所有文档进行词干提取和词频统计，这里是将数据包括训练集、验证集、测试集全部进行统计，由于实验数据量过大，提取词干和统计词频所耗费的时间会很多，为了节省时间，将所有统计结果保存到文件中，以便用之后的每次实验。

运用『TextBlob』进行词干提取得到一个list，再用『collections.Counter』对list的每项进行计数，存储到一个字典中，键值为 文件名，对应的值为词频结果。统计结果保存到『data/tf.txt』。

实现文件：『src/utils.py』

模型训练

分为三种模型，训练过程是相同的：

1. 计算每一类文本出现的概率 $p(d_i) = \frac{c(d_i)}{c(d)}$
2. 建立字典，确立一个大的空间，保证每一类文档的特征都是相同的
3. 统计每一类文档对应的词频,这里 伯努利模型与多项式模型、混合模型是不同的，伯努利模型不重复统计同一文档中多次出现的词。
4. 再将计算特征 -- 词的概率 $p(w|d_i) = \frac{c(w)}{c(w_{d_i total})}$

至此模型训练完毕。

实现文件：『src/Model.py』

测试模型

在测试模型中会遇到如下问题，测试集中有的词没有在训练好的模型中，这样会导致这个词的概率为0，从而影响模型分类的结果，这里用到了一个小 trick -- 平滑技术。平滑技术分为两种：

- 对于多项式模型，$p(w|d_i) = \frac{c(w)+1}{c(w_{d_i total})+c(w_{total})}$
- 对于伯努利模型，$p(w|d_i) = \frac{c(w)+1}{c(w_{d_i total})+c(d)}$

这样，即使$c(w)$为0也不会使预测概率为0。由于每一项的概率很小，大概是 10e-3 级别，所以为了方便运算，对每一项都取对数，简化计算。由贝叶斯定理可得：

$p(d_i|x_1, x_2 \ldots . x_n) = \frac{p(x_1,x_2,\ldots..x_n|d_i)*p(d_i)}{p(x_1,x_2,\ldots..x_n)}$

而由于对于同一文档，$p(x_1, x_2, \ldots . x_n)$是相同的，所以我们只需要计算$p(x_1, x_2, \ldots . x_n|d_i) * p(d_i)$即可，对该式子取对数得到 $\sum logp(x_i|d_i) + logp(d_i)$。

遇到的问题

一开始我采用的是对每一类文档提取一次字典，正确率只有 3%左右。更换成全局字典之后，效果提升非常明显...

实验结果

实验结果保存到 『../log/logout.log』中

多项式模型结果

> PolynomialModel Test Finished cost time 23.390124082565308s, Predict accuary is 6232 / 7517, 82.90541439404018%

伯努利模型结果

> BernoulliModel Test Finished cost time 26.7016499042511s, Predict accuary is 6445 / 7517, 85.73899161899693%

混合模型结果

> MixModel Test Finished cost time 31.27314591407776s, Predict accuary is 6192 / 7517, 82.37328721564454%

3. sklearn 聚类算法

项目地址

运行代码

```
python src/main.py -f data/Tweets.txt -l log/logfile.log
```

实验环境与实验数据

- CPU -- i7-8700K
- 内存 -- 15.6 GB
- 实验数据: Tweets.txt

实验目的

熟悉 sklearn 库中的常用聚类算法 API, 对 Tweets 数据进行聚类, 并比较效果.

实验步骤

读取数据

对 『Tweets.txt 』中的数据进行读取, 每一条数据都是是 『JSON 』格式的,所以通过 Python 的 『json』 模块对 『Tweets.txt』 中数据的每一行进行读取.

实现文件: 『src/main.py』

聚类及评估

通过 sklearn 库中的 API, 对数据进行聚类.运用了如下几个方法:

- KMeans
- AffinityPropagation
- MeanShift
- SpectralClustering
- Ward Hierarchical Clustering
- AgglomerativeClustering
- DBSCAN
- GaussianMixture

进行聚类.通过 NMI 对聚类结果进行评价.

$$I(A,B) = H(A) + H(B) - H(A,B) \qquad (1)$$

若用 $P_A(a)$、$P_b(b)$ 表示 A、B 的概率分布，$P_{AB}(a, b)$ 表示 A 和 B 的联合概率分布[11]，则：

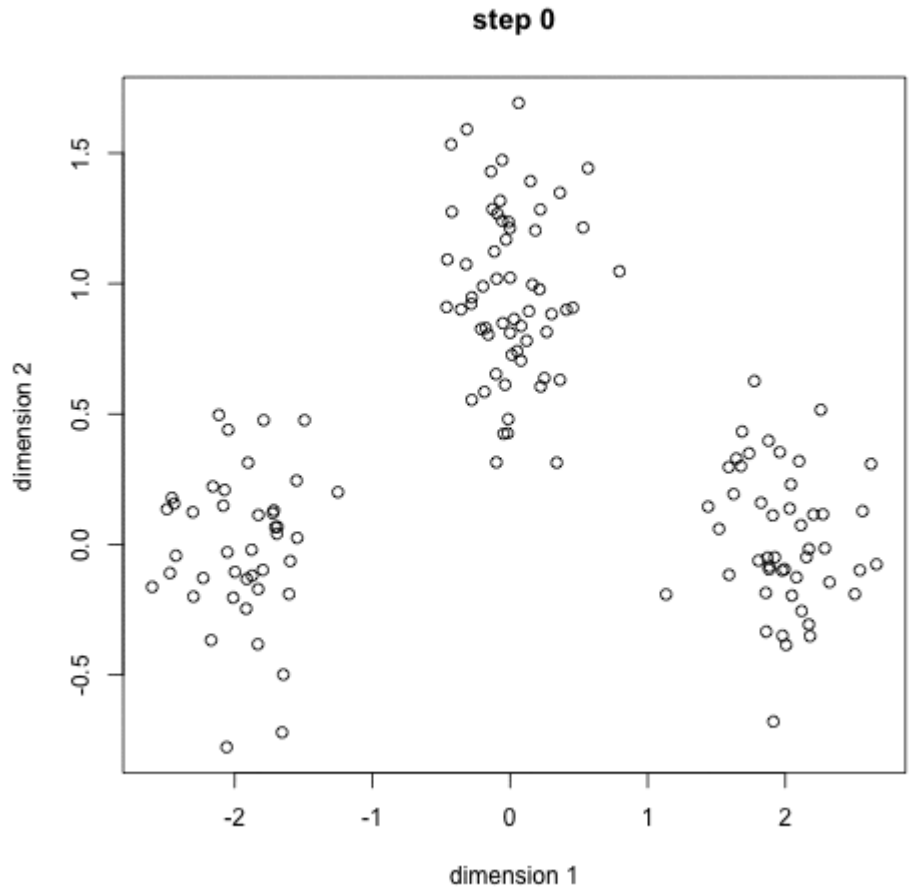
$$H(A) = -\sum_a P_A(a) \log P_A(a) \qquad (2)$$

$$H(B) = -\sum_a P_B(b) \log P_B(b) \qquad (3)$$

$$H(A,B) = -\sum_{a,b} P_{AB}(a,b) \log P_{AB}(a,b) \qquad (4)$$

其中I(A,B)是A,B两向量的mutual information, H(A)是A向量的信息熵。

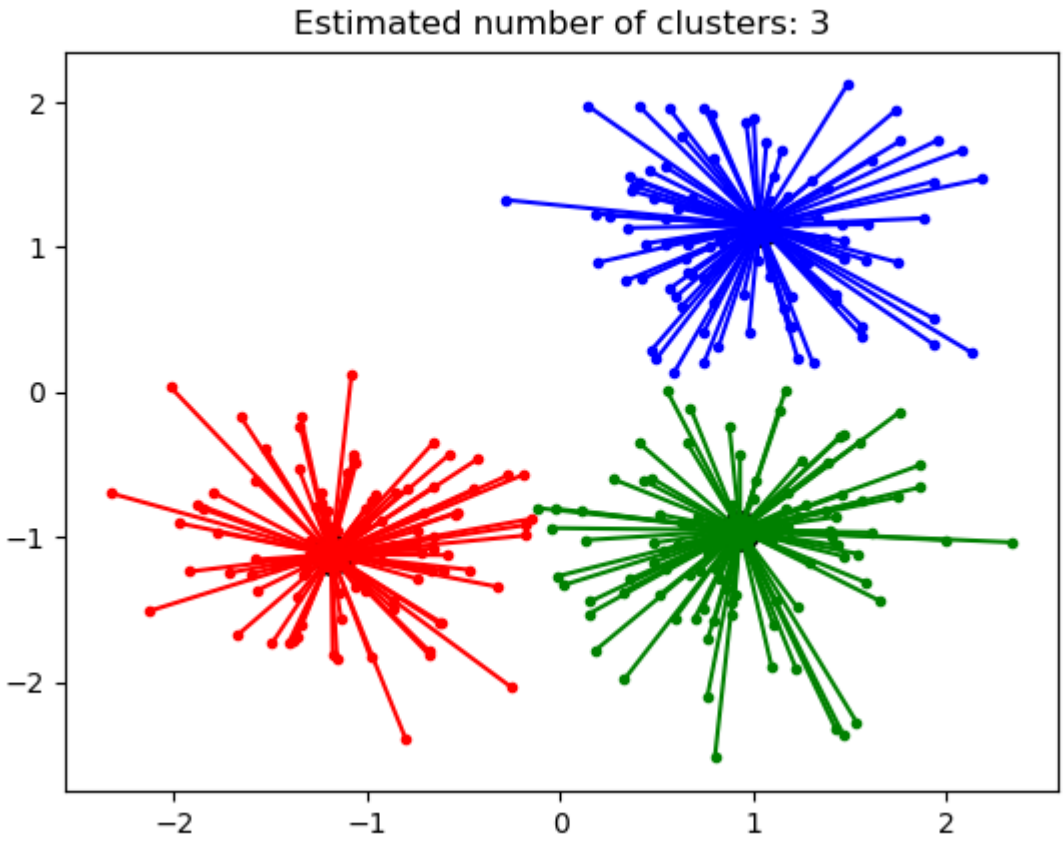
1. KMeans



1. 首先，选择一些类/组来使用并随机地初始化它们各自的中心点。
2. 每个数据点通过计算点和每个组中心之间的距离进行分类，然后将这个点分类为最接近它的组。
3. 基于这些分类点，我们通过取组中所有向量的均值来重新计算组中心。
4. 对一组迭代重复这些步骤。

结果: KMeans cost time 38.968318462371826s, score is 0.781562372454415

2. AffinityPropagation



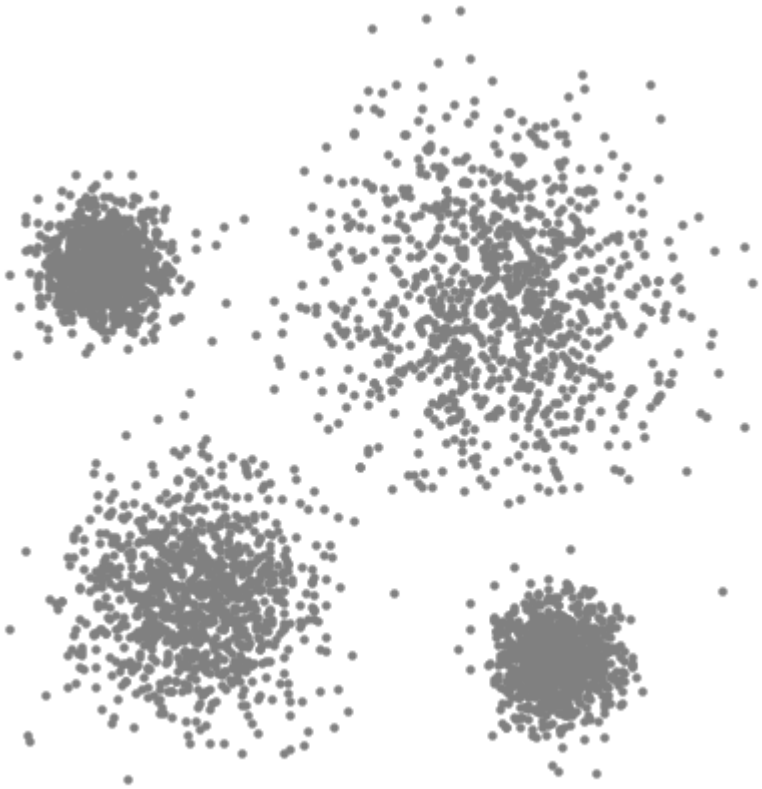
AP算法的基本思想是将全部样本看作网络的节点，然后通过网络中各条边的消息传递计算出各样本的聚类中心。聚类过程中，共有两种消息在各节点间传递，分别是吸引度(responsibility)和归属度(availability) 。A

P算法通过迭代过程不断更新每一个点的吸引度和归属度值，直到产生m个高质量的Exemplar（类似于质心），同时将其余的数据点分配到相应的聚类中。

结果: AffinityPropagation cost time 13.451494693756104s, score is 0.7836988975391974

3. MeanShift

均值偏移（Mean shift）聚类算法是一种基于滑动窗口（sliding-window）的算法，它试图找到密集的数据点。而且，它还是一种基于中心的算法，它的目标是定位每一组群/类的中心点，通过更新中心点的候选点来实现滑动窗口中的点的平均值。这些候选窗口在后期处理阶段被过滤，以消除几乎重复的部分，形成最后一组中心点及其对应的组。

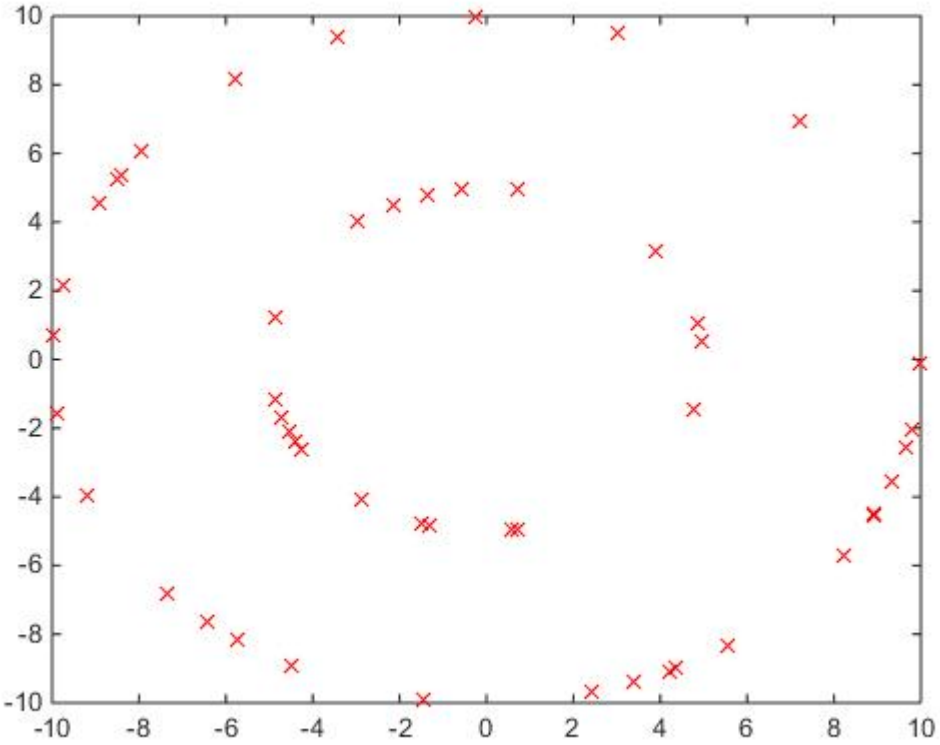


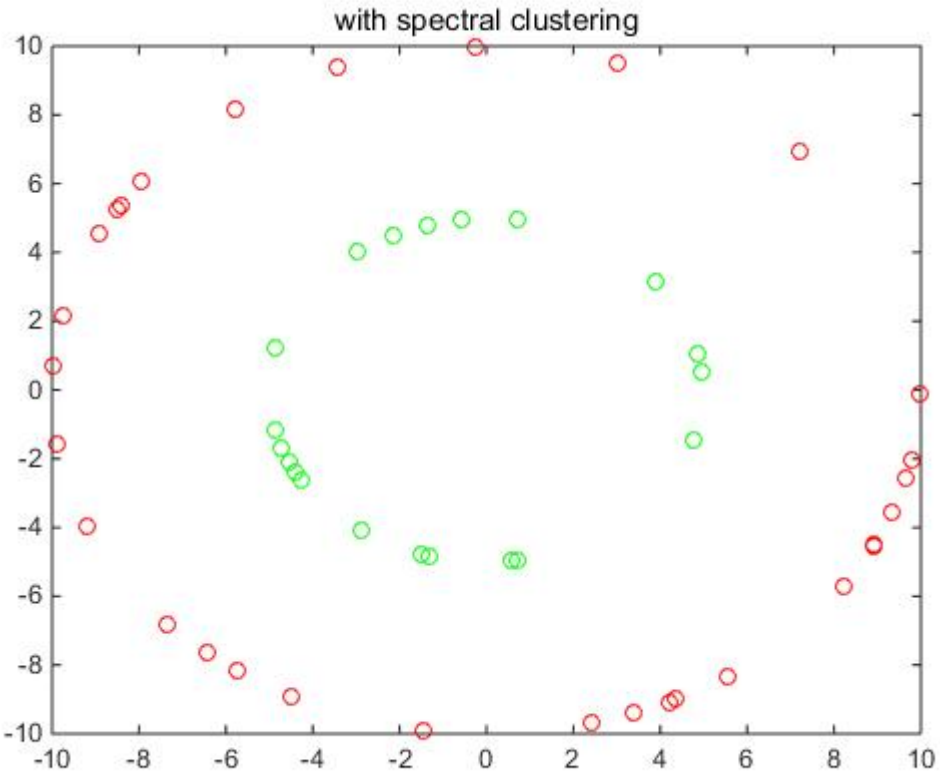
与K-Means聚类相比，均值偏移不需要选择聚类的数量，因为它会自动地发现这一点。这是一个巨大的优势。聚类中心收敛于最大密度点的事实也是非常可取的，因为它非常直观地理解并适合于一种自然数据驱动。

缺点是选择窗口大小/半径r是非常关键的，所以不能疏忽。

结果: MeanShift cost time 19.27985906600952s, score is 0.7278545708737196

4. SpectralClustering





1. 根据输入的相似矩阵的生成方式构建样本的相似矩阵S
2. 根据相似矩阵S构建邻接矩阵W，构建度矩阵D
3. 计算出拉普拉斯矩阵L
4. 构建标准化后的拉普拉斯矩阵
5. 计算拉普拉斯矩阵最小的k_1 个特征值所各自对应的特征向量

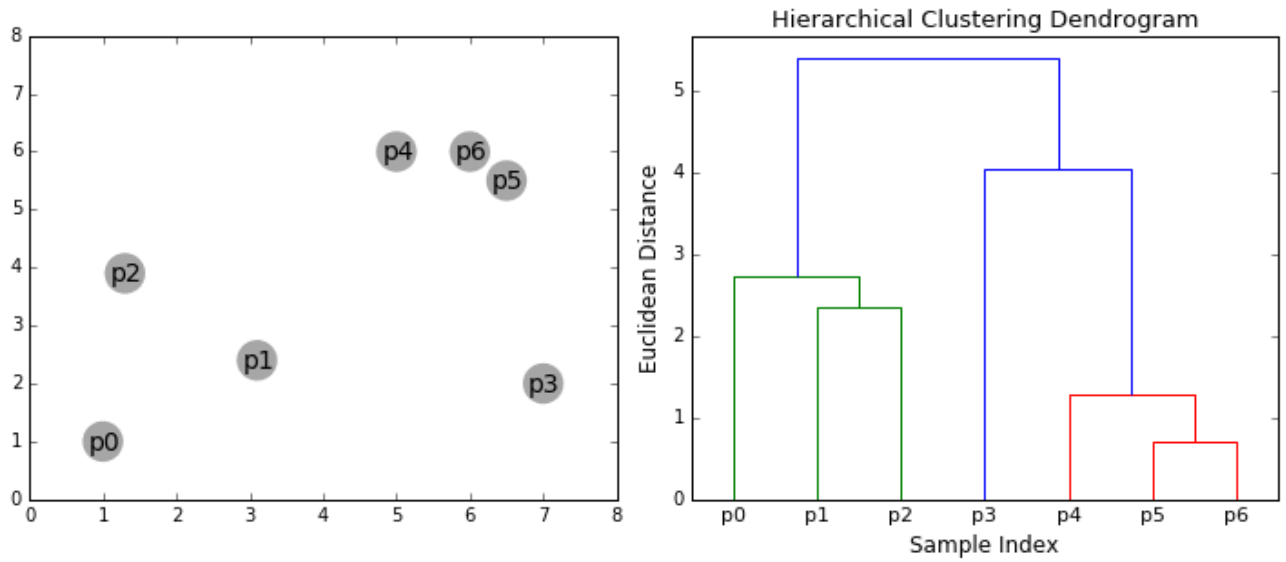
6. 将各自对应的特征向量ff组成的矩阵按行标准化，最终组成$n * k_1$维的特征矩阵F
7. 对F中的每一行作为一个k_1维的样本，共n个样本，用输入的聚类方法进行聚类，聚类维数为k_2。
8. 得到簇划分C(c1,c2,...ck2).

结果: SpectralClustering cost time 40.73340439796448s, score is 0.8067027949127966

5. AgglomerativeClustering

层次聚类（hierarchical clustering）可在不同层次上对数据集进行划分，形成树状的聚类结构。AggregativeClustering是一种常用的层次聚类算法。

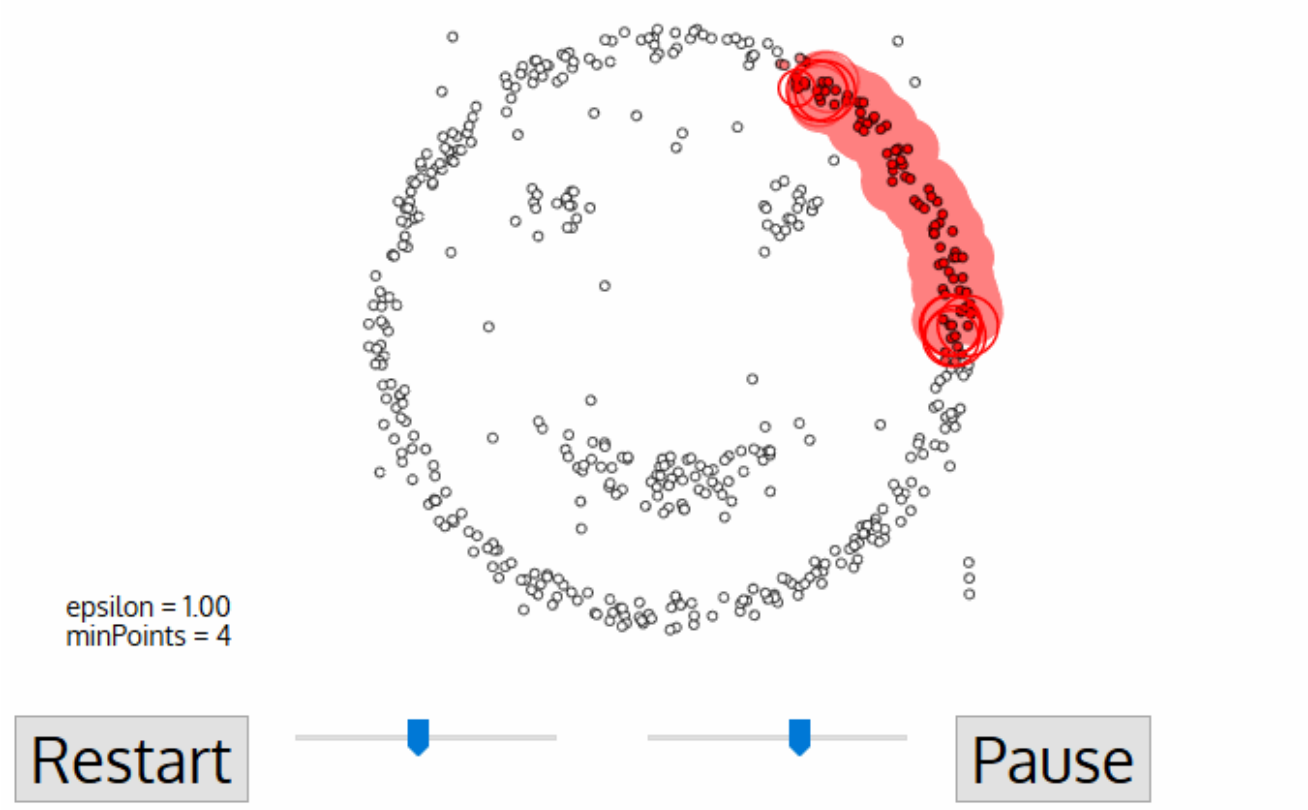
其原理是：最初将每个对象看成一个簇，然后将这些簇根据某种规则被一步步合并，就这样不断合并直到达到预设的簇类个数。



1. 我们首先将每个数据点作为一个单独的聚类进行处理。
2. 在每次迭代中，我们将两个聚类合并为一个。
3. 重复步骤2直到我们到达树的根。层次聚类算法不要求我们指定聚类的数量，我们甚至可以选择哪个聚类看起来最好。此外，该算法对距离度量的选择不敏感.

结果: AgglomerativeClustering cost time 7.590268611907959s, score is 0.7847178748775534

6. DBSCAN



DBSCAN(Density-Based Spatial Clustering of Applications with Noise)是一个比较有代表性的基于密度的聚类算法，类似于均值转移聚类算法.

1. DBSCAN以一个从未访问过的任意起始数据点开始。
2. 如果在这个邻域中有足够数量的点（根据 minPoints），那么聚类过程就开始了，并且当前的数据点成为新聚类中的第一个点。否则，该点将被标记为噪声（稍后这个噪声点可能会成为聚类的一部分）。在这两种情况下，这一点都被标记为“访问（visited）”。
3. 对于新聚类中的第一个点，其ε距离附近的点也会成为同一聚类的一部分。
4. 步骤2和步骤3的过程将重复，直到聚类中的所有点都被确定，就是说在聚类附近的所有点都已被访问和标记。
5. 一旦我们完成了当前的聚类，就会检索并处理一个新的未访问点，这将导致进一步的聚类或噪声的发现。这个过程不断地重复，直到所有的点被标记为访问。

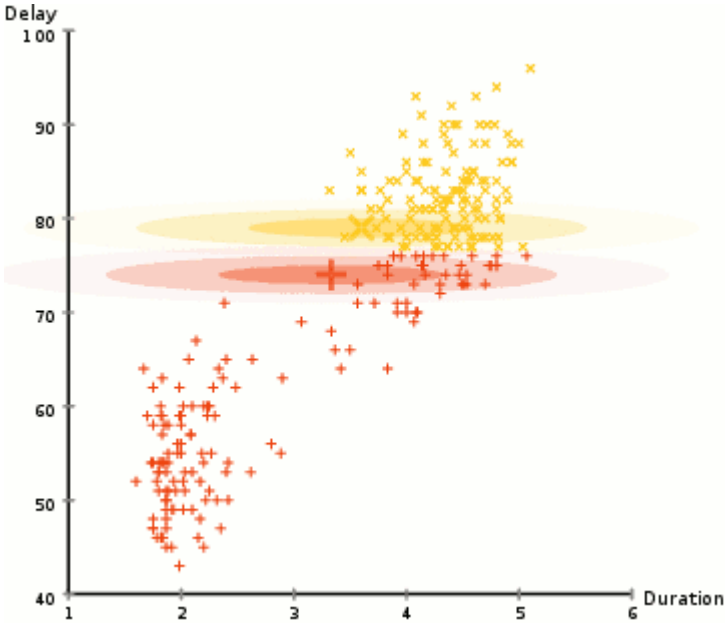
结果: DBSCAN cost time 46.81977677345276s, score is 0.611995415736244

7. GaussianMixture

K-Means的一个主要缺点是它对聚类中心的平均值的使用很简单。

高斯混合模型（GMMs）比K-Means更具灵活性。使用高斯混合模型，我们可以假设数据点是高斯分布的;比起说它们是循环的，这是一个不那么严格的假设。这样，我们就有两个参数来描述聚类的形状：平均值和标准差以二维的例子为例，这意味着聚类可以采用任何形式的椭圆形状（因为在x和y方向上都有标准差）。因此，每个高斯分布可归属于一个单独的聚类。

为了找到每个聚类的高斯分布的参数（例如平均值和标准差）我们将使用一种叫做期望最大化（EM）的优化算法。看看下面的图表，就可以看到高斯混合模型是被拟合到聚类上的。然后，我们可以继续进行期望的过程——使用高斯混合模型实现最大化聚类。



1. 我们首先选择聚类的数量（如K-Means所做的那样），然后随机初始化每个聚类的高斯分布参数。
2. 给定每个聚类的高斯分布，计算每个数据点属于特定聚类的概率。
3. 基于这些概率，我们为高斯分布计算一组新的参数，这样我们就能最大程度地利用聚类中的数据点的概率。
4. 步骤2和3被迭代地重复，直到收敛。

结果: GaussianMixture cost time 5.616118431091309s, score is 0.7834852894647376

实验结果

需要对聚类方法的效果以及时间 cost 进行综合评估, 发现虽然 SpectralClustering 的方法准确率是最高的,但是耗费时间也是最长的.综合看下来 GaussianMixture 的效果最好.