**信 息 内 容 安 全 实 验 报 告**

|  |  |
| --- | --- |
| 实验项目名称： | 话题检测 |
| 班级： | SC011701 |
| 学号： | 2017302207 |
| 姓名： | 高丽 |
| 指导教师： | 杨黎斌 |
| 实验时间： | 2020.4.9 |

目录

[1. 总体概述 1](#_Toc37447307)

[2. 原理 1](#_Toc37447308)

[2.1 主题词提取——TF-IDF 1](#_Toc37447309)

[2.2 K-Means算法 3](#_Toc37447310)

[2.2.1 算法流程 3](#_Toc37447311)

[2.2.2 距离计算和准则函数 3](#_Toc37447312)

[2.2.3 标签映射——Kuhn-Munkres算法 4](#_Toc37447313)

[2.2.4 评估方法 7](#_Toc37447314)

[3 K-Means变种 10](#_Toc37447315)

[3.1 K-Means++ 10](#_Toc37447316)

[3.2 二分K-Means 11](#_Toc37447317)

[4 实现和代码 12](#_Toc37447318)

[4.1 总体流程 12](#_Toc37447319)

[4.2 预处理和特征提取 12](#_Toc37447320)

[4.3 K-Means 15](#_Toc37447321)

[4.4 K-Means++ 18](#_Toc37447322)

[4.5 二分K-Means 19](#_Toc37447323)

[4.6 标签匹配和评估指标 21](#_Toc37447324)

[4.6.1 KM算法、纯度计算代码 21](#_Toc37447325)

[4.6.2 其他指标评估代码 22](#_Toc37447326)

[4.7 预测部分代码 24](#_Toc37447327)

[5 实验结果和分析 26](#_Toc37447328)

[5.1 直观显示 26](#_Toc37447329)

[5.2 指标评估 27](#_Toc37447330)

[5.3 原因分析 28](#_Toc37447331)

# 1. 总体概述

首先,针对给定的 3445个文件，考虑到K-Means运行效率和特征的问题，不选择太多的文件作为训练集，所以我随机选了每个C中20个文件作为训练集，共120个；

随机选择每个C中10个文件作为测试集（若不足10个则选择该类所有剩余的文件），利用TF-IDF对训练集中每个样本筛选出特征词，构建特征空间，利用TF-IDF值为每个文档构建词向量；

对文档聚类：设定k值为6，分别使用二分KMeans、KMeans++算法。

给出聚类评估方法——已知标签情况下的纯度、互信息分和兰德系数，以及真实标签未知情况下的卡林斯基-哈拉巴斯指数、轮廓系数。

利用Kuhn-Munkres算法实现了聚类结果中预测标签到原始真实标签的映射，方便后续判断准确率；

输入测试集，计算某个文件对应向量到各个质心的距离，根据距离最小原则预测分类，这一步用到了前面的标签映射，同时进一步判断分类的准确率。

**实验为个人独立完成**。

# 2. 原理

## 2.1 主题词提取——TF-IDF

一般来说，一篇文章的关键词是文章的高频词，但是有些词汇是普遍常用的，尽管词频较高，却无法作为文章的主旨词汇。所以我们引入TF-IDF对这一类词进行过滤：

如果某个词比较少见，但是它在这篇文章中多次出现，那么它很可能就反映了这篇文章的特性，这里正是我们所需要的关键词。

首先统计指定文档的词频（Term Frequency，缩写为TF）。

考虑到如果某个词比较少见，但是它在这篇文章中多次出现，那么其重要性会偏高。

在词频的基础上，要对每个词分配一个“重要性”权重。较常见的词给予较小的权重，较少见的词给予较大的权重。这个权重叫做"逆文档频率"（Inverse Document Frequency，缩写为IDF），它的大小与一个词的常见程度成反比。

某个词对文章的重要性越高，它的TF-IDF值就越大。所以，排在最前面的几个词，一般就是这篇文章的关键词。

在预处理过程中，首先进行特征提取，筛选每篇文章的主题词，选取所有主题词作为特征，用tf-idf值构建每篇文章的特征向量，在筛选过程中，设定两个参数min\_df、max\_df即最小文档频率和最大文档频率：

**max\_df**:有些词，它们的文档频率太高了，一个词如果每篇文档都出现，那就没有必要用它来区分文本类别了），所以，可以设定一个阈值，比如float类型0.5（取值范围[0.0 , 1.0]），表示这个词如果在整个数据集中超过50%的文本都出现了，那么就把它列为临时停用词。也可以设定为int型，例如max\_df=10，表示这个词如果在整个数据集中超过10的文本都出现了，那么也把它列为临时停用词。

**min\_df**:与max\_df相反，虽然文档频率越低，似乎越能区分文本，可是如果太低，例如10000篇文本中只有一篇文本出现过这个词，仅仅因为这一篇文本，就增加了词向量空间的维度，不太划算。所以小于min\_df的词也应当去除。

以上，通过筛选出min\_df和max\_df之间的词，得出所有特征（词）。

## 2.2 K-Means算法

### 2.2.1 算法流程

KMeans算法将一组N个样本的特征矩阵X划分为K个无交集的簇，直观上来看是簇是一组一组聚集在一起的数据，在一个簇中的数据就认为是同一类。簇就是聚类的结果表现。

簇中所有数据的均值通常被称为这个簇的“质心”（centroids）。

在KMeans算法中，簇的个数K是一个超参数，需要我们人为输入来确定。KMeans的核心任务就是根据我们设定好 的K，找出K个优的质心，并将离这些质心近的数据分别分配到这些质心代表的簇中去。具体过程可以总结如下：

1、随机抽取K个样本作为初的质心

2、开始循环：

将每个样本点分配到离他们近的质心，生成K个簇

对于每个簇，计算所有被分到该簇的样本点的平均值作为新的质心

3、当质心的位置不再发生变化，迭代停止，聚类完成。

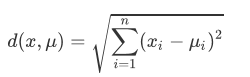
### 2.2.2 距离计算和准则函数

距离计算：

聚类算法的目的是追求“簇内差异小，簇外差异大”。而这个“差异”，由样本点到其所在簇的质心的距离来衡量。

对于一个簇来说，所有样本点到质心的距离之和越小，我们就认为这个簇中的样本越相似，簇内差异就越小。而距离的衡量方法有多种，表示簇中的一个样本点， 表示该簇中的质心，n表示每个样本点中的特征数目，i表示组成点的每个特征，则该样本点到质心的距离可以由以下距离来度量：

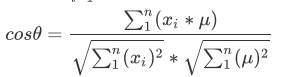
①欧几里得距离：



②曼哈顿距离：



③余弦距离：



本次实验均采用欧式距离计算。

准则函数：

KMeans算法不断重复迭代过程，使得准则函数收敛，通常准则函数为平方误差和准则函数，即SSE（sum of the quared error），其定义如下：

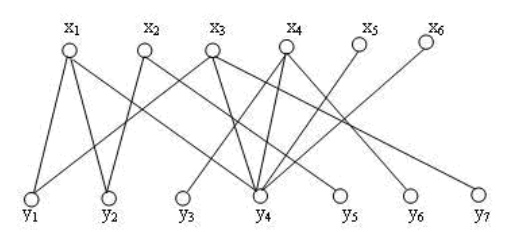
SSE是数据库中所有对象的平方和误差总和，x为数据对象，p为簇的平均值。

### 2.2.3 标签映射——Kuhn-Munkres算法

假设{1,1,1,2,2,3,4,4,⋯,7,7,7,9,9}为真实的数据标签，而利用上述算法对数据聚类之后的标签可能是{3,3,3,1,1,4,6,6,⋯,9,9,7,7,7}，就算在最理想的状态下，把所有该在一个簇中的数据都聚类到一个簇中，但是由于标签的规则不一致，所以每个簇给定的标号不一样。这时候就需要利用到Kuhn-Munkres算法来对标签进行重新标号。

在介绍Kuhn-Munkres算法思想前，需要提到匈牙利算法：

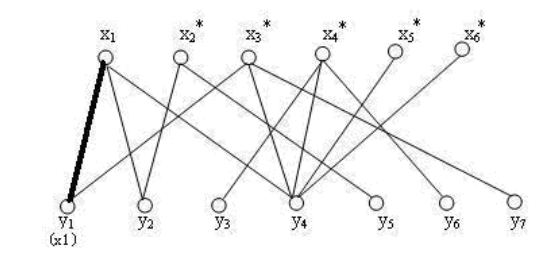
给定一个二部图：



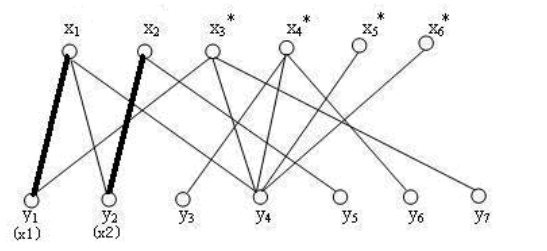
算法目标是尽可能给x中最多的点找到配对。

最大匹配是互相的，如果我们给X找到了最多的Y中的对应点，同样，Y中也不可能有更多的点得到匹配了。

刚开始，一个匹配都没有，我们随意选取一条边，（x1, y1）这条边，构建最初的匹配出来，结果如下，已经配对的边用粗线标出：

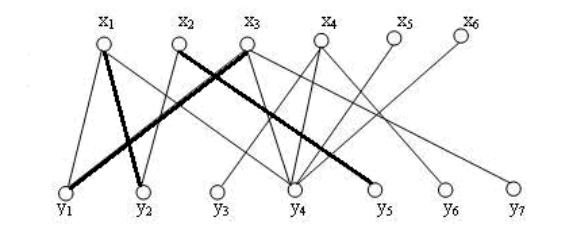


我们给x2添加一个匹配，如下图的（x2, y2）边:



到这里，我们形成了匹配M，其有（x1, y1）, (x2, y2 ) 两条边。

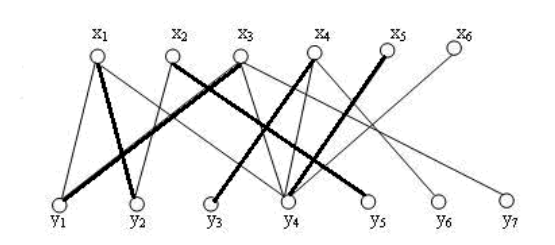
我们现在想给x3匹配一条边，发现它的另一端y1已经被x1占用了，那x3就会追溯到y1，y1追溯到已经匹配的x1。x1失去匹配，寻找y2，而y2被x2占用，则使x2和y2分开；x2寻找y5，y5未被占用，于是x2和y5成功匹配：



上述过程是一个迭代的过程，牵涉到的节点（x3, y1, x1, y2, x2, y5）形成了一条路径P，该路径是之前的匹配M的一条增广路径。

发现一条增广路径，就意味着一个更大匹配的出现，于是，我们将M中的配对点拆分开，重新组合，得到了一个更大匹配，M1, 其拥有（x3, y1）,(x1, y2), (x2, y5)三条边。

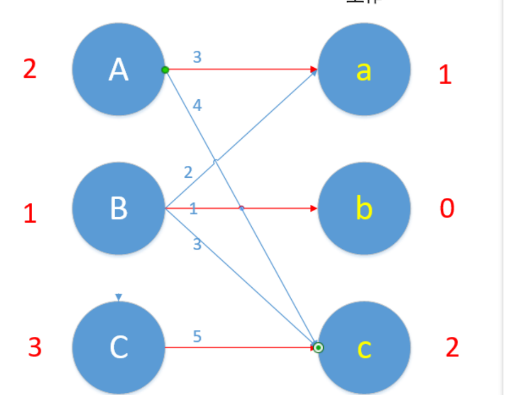
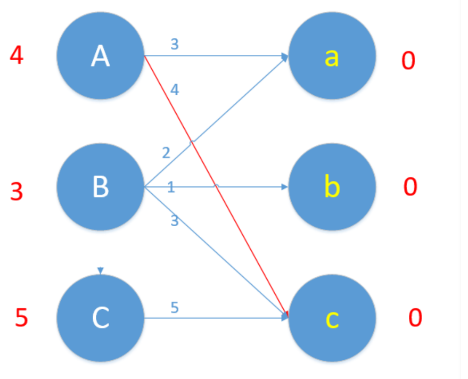
同样，x4 , x5 按顺序加入进来，最终会得到本图的最大匹配。



匈牙利算法是求解二部图最大匹配的算法，而KM算法是求解赋权二部图最大匹配的算法：

之前说到重新标号——主要思路就是，将聚类之后的每一种标签（称为预测标签）与每一种真实标签一一对比，计算出所有标签组合（i，j）情况的数据重合数量，这样就会形成一个“代价矩阵”，基于这个“代价矩阵”，就可以利用Kuhn-Munkres算法来找出代价最低的分配方式，也就是预测标签与真实标签之间的映射关系。

赋权二分图中，点与点之间的边带有权值，对于点来说，它的顶标是所有相连边权值中最大的权值。我们可以把它理解为期望值。权值可以理解为选择的期望值。

**

Kuhn-Munkres算法根本上是遵从**+，**也就是顶标相加大于权值的原则。通过不断地迭代实现整个二分图最大权值匹配。

其中为的顶标。

Kuhn-Munkres算法的流程如下：

在赋权完全二部图G中，从G的任何一可行顶点标号l开始，求出相等子图；

在中执行匈牙利算法，如果求得的的一个完美匹配M时，输出M，算法停止，否则，匈牙利算法必将终止于两个集合且 ，此时转入下一步；

计算G的可行顶点标号，并代替,代替，转上一步。

### 2.2.4 评估方法

目前，比较常用的评价聚类质量的指标主要有外部和内部聚类有效性。外部聚类有效性是指利用聚类分析算法对数据集进行处理，所得到的聚类划分结果与数据集中的“真实”聚类划分结果进行比较，来对其有效性进行评估。内部聚类有效性是主要考虑数据集自身的几何分布特征来对聚类结果进行评价。

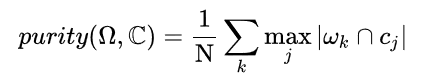
#### 当真实标签已知时

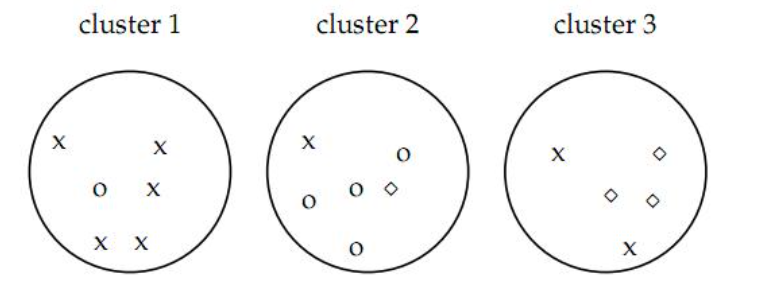
虽然在聚类中不输入真实标签，但这不代表数据中一定不具有真实标签，或者一定没有任何参考信息。

如果有样本真实聚类情况的数据，我们可以对于聚类算法的结果和真实结果来衡量聚类的效果。常用的有纯度、互信息分、同质性、完整性、兰德系数等，这里给出互信息和兰德系数的介绍：

**① 纯度（purity）/准确率（accuracy）**

对于本次实验，最快速评估的方法是计算纯度（Purity），也就是，正确聚类的文档数占总文档的比例（准确率），我们把每个簇中最多的类作为这个簇所代表的类，然后计算正确分配的类的数量，然后除以N。形式化表达如下：





例如上面的例子，purity= ( 5+4+3) / 17 = 0.71 第一个cluster正确的有5个，第二个cluster正确的有4个，第3个cluster正确的有3个

这个方法计算简单，值在0~1之间，完全错误值为0，完全正确值为1。

根据之前标签映射的结果，可以很快的求出簇的纯度。

当簇的数量很多的时候，容易达到较高的纯度——特别是，如果每个文档都被分到独立的一个簇中，那么计算得到的纯度就会是1。因此，不能简单用纯度来衡量聚类质量与聚类数量之间的关系。

**② 互信息分**

互信息分可以解决简单计算纯度带来的弊端，它用来衡量两个数据分布的吻合程度。假设U与V是对N个样本标签的分配情况，则两种分布的熵（熵表示的是不确定程度）分别为

其中，

U与V之间的互信息MI定义为：

其中，

标准化后的互信息（Normalized mutual information）为：

与ARI类似，调整互信息（Adjusted mutual information）定义为：

利用基于互信息的方法来衡量聚类效果需要实际类别信息，MI与NMI取值范围为[0,1]，AMI取值范围为[−1,1]，它们都是值越大意味着聚类结果与真实情况越吻合。

**③ 兰德指数（Rand index）**

需要给定实际类别信息C，假设K是聚类结果，a表示在C与K中都是同类别的元素对数，b表示在C与K中都是不同类别的元素对数，则兰德指数为：

其中数据集中可以组成的总元素对数，RI取值范围为[0,1]，值越大意味着聚类结果与真实情况越吻合。

对于随机结果，RI并不能保证分数接近零。为了实现“在聚类结果随机产生的情况下，指标应该接近零”，调整兰德系数（Adjusted rand index）被提出，它具有更高的区分度：

ARI取值范围为[−1,1]，值越大意味着聚类结果与真实情况越吻合。从广义的角度来讲，ARI衡量的是两个数据分布的吻合程度。

#### 当真实标签未知时

最常用的有轮廓系数和卡林斯基-哈拉巴斯指数（CHI）等，这里介绍轮廓系数：

**轮廓系数**适用于实际类别信息未知的情况。对于单个样本，设a是与它同类别中其他样本的平均距离，b是与它距离最近不同类别中样本的平均距离，轮廓系数为：

对于一个样本集合，它的轮廓系数是所有样本轮廓系数的平均值。

轮廓系数取值范围是[−1,1]，同类别样本越距离相近且不同类别样本距离越远，分数越高。

如果一个簇中的大多数样本具有比较高的轮廓系数，则簇会有较高的总轮廓系数，则整个数据集的平均轮廓系数越高，则聚类是合适的。如果许多样本点具有低轮廓系数甚至负值，则聚类是不合适的，聚类的超参数K可能设定得太大或者太小。

# 3 K-Means变种

## 3.1 K-Means++

K-Means++算法选择初始seeds的基本思想就是：初始的聚类中心之间的相互距离要尽可能的远。该算法的描述是如下:

1、从输入的数据点集合中随机选择一个点作为第一个聚类中心

2、对于数据集中的每一个点x，计算它与最近聚类中心(指已选择的聚类中心)的距离D(x)

3、选择一个新的数据点作为新的聚类中心，选择的原则是：D(x)较大的点，被选取作为聚类中心的概率较大

4、重复2和3直到k个聚类中心被选出来

5、利用这k个初始的聚类中心来运行标准的k-means算法

在上述第三步中：

①先从我们的数据库随机挑个随机点当“种子点”  
②对于每个点，我们都计算其和最近的一个“种子点”的距离D(x)并保存在一个数组里，然后把这些距离加起来得到Sum(D(x))。  
③然后，再取一个随机值，用权重的方式来取计算下一个“种子点”。这个算法的实现是，先取一个能落在Sum(D(x))中的随机值Random，然后用Random -= D(x)，直到其<=0，此时的点就是下一个“种子点”。  
④重复2和3直到k个聚类中心被选出来  
⑤利用这k个初始的聚类中心来运行标准的k-means算法

## 3.2 二分K-Means

为了克服K-Means算法收敛于局部最小值的问题，提出了一种二分K-均值(bisecting K-means)，该算法首先将所有点作为一个簇，将其一分为二，之后选择其中一个簇继续进行划分，选择哪一个簇进行划分，取决于对其划分是否可以最大程度降低SSE的值。上述基于SSE的划分过程不断重复，直到得到用户指定的簇数目为止：  
 算法的伪代码如下：

将所有的点看成是一个簇

当簇小于数目k时

对于每一个簇

计算总误差

在给定的簇上进行K-均值聚类,k值为2

计算将该簇划分成两个簇后总误差

选择使得误差最小的那个簇进行划分

# 4 实现和代码

## 4.1 总体流程

（1）文件转换编码全部统一为utf-8；（convertformat.py）

（2）随机选择每个类20个文件放入test文件夹中，每个类10个文件放入train文件夹中，然后预处理：（init\_sets.py）

①读取train每个文件，利用哈工大的nytlp包进行分词、去标点符号和词性选择（只选择动词和名词）；（preprocess.py）

②对于每篇文章的所有词语计算TF-IDF值，根据设定的min\_df、max\_df筛选掉在少数文档中出现的词语和在大多数文档中出现的词语，利用TF-IDF值构建每个文档的特征向量；（my\_kmeans.py中）

（3）利用多种K-Means算法对训练集聚类，并且给出评估指标——纯度、互信息分、兰德系数、轮廓系数；（my\_kmeans.py中）

（4）将聚类好的标签映射到原始的真实标签；利用训练好的模型去检测test下的文件的分类，并计算正确率。（my\_kmeans.py中）

画图部分利用PCA降维绘图。

## 4.2 预处理和特征提取

预处理部分主要工作：

* 分词
* 词性标注和选择，我选择的是动词和名词，这两种词性的词更能代表文章内容
* 去标点符号
* 计算TF-IDF，构建特征向量，根据max\_df和min\_df筛选词特征，最构建好的特征矩阵进行正则化处理

分词和词性筛选部分代码：

def get\_allfile\_content(filelist,path,posts):

#posts为可以设定的词性列表

#根据posts筛选出需要的某些词性的词语

allf\_content=[]#所有文件的分词结果 类型：list of str

segmentor=Segmentor()# # 初始化分词模型实例

segmentor.load(cws\_model\_path)#根据地址加载模型

if posts!=[]:#如果词性列表不为空

postagger = Postagger() # 初始化词性标注模型实例

postagger.load(pos\_model\_path) # 加载模型

count=0#计数

for i in filelist:#i为单个文件名

punctuations = """,，.。：…・《》［］～[]【】－－〈〉―()②③④⑤⑥⑦⑧⑨⑩①①①②①④①⑦（）/~\*&^%$#@＠！“”‘；~`[]{|、}\\/\n\t\r~+\_-=？"""

digit='01234567897一二三四五六七八九０１２３４５６７８９'

alpha='ａabcdefghijklmnopqrstuvwxyz\_ABCDEFGHIJKLMNOPQRSTUVWXYZ'

punctuations+=digit

punctuations+=alpha

content=readfile(i,path)

for punc in punctuations:#去标点符号和特殊符号、数字

content=content.replace(punc,' ')

words=list(segmentor.segment(content))#分词

if posts!=[]:

postags = postagger.postag(words) # 词性标注

#选择需要的词性

newlist=[]

for a in posts:

for i in range(len(list(postags))):

if list(postags)[i][0]==a:#元素类型为str

newlist.append(words[i])

else:#不筛选词性

newlist=words

contentoffile=' '.join(newlist)#结果连接为字符串

allf\_content.append(contentoffile)

print(count)#进度显示

count+=1

segmentor.release()#释放模型

if posts!=[]:

postagger.release() # 释放模型

return allf\_content

特征提取部分代码：

#训练集

with open(config.allcorpuspath,'rb')as f:

model=pickle.load(f)#调用之前分好词的语料库（训练集）

sample=model['filecontent']#获取样本

print(len(model['filecontent']),'个')#输出样本个数

# #获取原始分类

filename=model['filename']

y\_true=[]#原始分类数组

#根据文件名字设定真实标签

for i in range(len(filename)):

if filename[i][:3]=='C4-':

y\_true.append(0)

elif filename[i][:3]=='C5-':

y\_true.append(1)

elif filename[i][:3]=='C7-':

y\_true.append(2)

elif filename[i][:3]=='C17':

y\_true.append(3)

elif filename[i][:3]=='C34':

y\_true.append(4)

elif filename[i][:3]=='C39':

y\_true.append(5)

else:

pass

#TF-IDF实例化 #token\_pattern保留长度为1的词

vec = TFIDF(token\_pattern=r"(?u)\b\w+\b",min\_df=0.05,max\_df =0.6,smooth\_idf=True)

X = vec.fit\_transform(sample)#训练样本

array1=X.toarray()#获取数组结果

array1=normalize(array1)#正则化

## 4.3 K-Means

这部分代码包含在源代码的Bikmeans.py中：

class KMeans(object):

def \_\_init\_\_(self,n\_clusters=5,initCent='random',max\_iter=300):

#initCent质心初始化方式，可选"random"或指定一个具体的array,默认random，即随机初始化

# max\_iter:最大迭代次数，可以更快停止算法

if hasattr(initCent, '\_\_array\_\_'):

n\_clusters = initCent.shape[0]#聚类个数，即k

self.centroids = np.asarray(initCent, dtype=np.float)

else:

self.centroids = None

self.n\_clusters = n\_clusters

self.max\_iter = max\_iter

self.initCent = initCent

self.clusterAssment = None

self.labels = None #标签

self.sse = None #SSE

#计算两点的欧式距离

def \_distEclud(self, vecA, vecB):

return np.linalg.norm(vecA - vecB)

#随机选取k个质心,必须在数据集的边界内

def \_randCent(self, X, k):

n = X.shape[1] #特征维数

centroids = np.empty((k,n)) #k\*n的矩阵，用于存储质心

for j in range(n): #产生k个质心，一维一维地随机初始化

minJ = min(X[:,j])

rangeJ = float(max(X[:,j]) - minJ)

centroids[:,j] = (minJ + rangeJ \* np.random.rand(k,1)).flatten()

return centroids

def fit(self, X):

#类型检查

if not isinstance(X,np.ndarray):

try:

X = np.asarray(X)

except:

raise TypeError("numpy.ndarray required for X")

m = X.shape[0]#m代表样本数量

self.clusterAssment = np.empty((m,2))#m\*2的矩阵，第一列存储样本点所属的族的索引值，

#第二列存储该点与所属族的质心的平方误差

if self.initCent == 'random':

self.centroids = self.\_randCent(X, self.n\_clusters)

clusterChanged = True

for \_ in range(self.max\_iter):

clusterChanged = False

for i in range(m):#将每个样本点分配到离它最近的质心所属的族

minDist = np.inf; minIndex = -1

for j in range(self.n\_clusters):

distJI = self.\_distEclud(self.centroids[j,:],X[i,:])

if distJI < minDist:

minDist = distJI; minIndex = j

if self.clusterAssment[i,0] != minIndex:

clusterChanged = True

self.clusterAssment[i,:] = minIndex,minDist\*\*2

if not clusterChanged:#若所有样本点所属的族都不改变,则已收敛，结束迭代

break

for i in range(self.n\_clusters):#更新质心，即将每个族中的点的均值作为质心

ptsInClust = X[np.nonzero(self.clusterAssment[:,0]==i)[0]]#取出属于第i个族的所有点

self.centroids[i,:] = np.mean(ptsInClust, axis=0)

self.labels = self.clusterAssment[:,0]

self.sse = sum(self.clusterAssment[:,1])

def predict(self,X):#根据聚类结果，预测新输入数据所属的族

if not isinstance(X,np.ndarray):

try:

X = np.asarray(X)

except:

raise TypeError("numpy.ndarray required for X")

m = X.shape[0]#m代表样本数量

preds = np.empty((m,))

for i in range(m):#将每个样本点分配到离它最近的质心所属的族

minDist = np.inf

for j in range(self.n\_clusters):

distJI = self.\_distEclud(self.centroids[j,:],X[i,:])

if distJI < minDist:

minDist = distJI

preds[i] = j

return preds

## 4.4 K-Means++

#K-Means++聚类部分

n\_clusters=6#类别数量

#K-Means++，slearn中默认使用K-Means++，实例化、训练

cluster=KMeans(n\_clusters=n\_clusters,random\_state=0).fit(array1)

y\_pred=cluster.labels\_#获取类别标签集合

#画图，不同的类别使用不同的颜色

color=['red','green','blue','pink','orange','gray']

fig,ax1=plt.subplots(1)#构建子图

for i in range(n\_clusters):

ax1.scatter(array1[y\_pred==i,0],array1[y\_pred==i,1],marker='o',s=8,c=color[i])

plt.show()

## 4.5 二分K-Means

#二分K-Means部分

class biKMeans(object):

def \_\_init\_\_(self,n\_clusters=6):

self.n\_clusters = n\_clusters#簇的个数

self.centroids = None#质心

self.clusterAssment = None#用于保存簇信息

self.labels = None#标签

self.sse = None#SSE

#计算两点的欧式距离

def \_distEclud(self, vecA, vecB):

return np.linalg.norm(vecA - vecB)

#训练

def fit(self,X):

m = X.shape[0]#样本数量

self.clusterAssment = np.zeros((m,2))

centroid0 = np.mean(X, axis=0).tolist()#初始质点

centList =[centroid0]

for j in range(m):#计算每个样本点与质心之间初始的平方误差

self.clusterAssment[j,1] = self.\_distEclud(np.asarray(centroid0), X[j,:])\*\*2

while (len(centList) < self.n\_clusters):

lowestSSE = np.inf

for i in range(len(centList)):#尝试划分每一族,选取使得误差最小的那个族进行划分

ptsInCurrCluster = X[np.nonzero(self.clusterAssment[:,0]==i)[0],:]

clf = KMeans(n\_clusters=2)#两个簇

clf.fit(ptsInCurrCluster)

#划分该族后，所得到的质心、分配结果及误差矩阵

centroidMat, splitClustAss = clf.centroids, clf.clusterAssment

sseSplit = sum(splitClustAss[:,1])

sseNotSplit = sum(self.clusterAssment[np.nonzero(self.clusterAssment[:,0]!=i)[0],1])

if (sseSplit + sseNotSplit) < lowestSSE:

bestCentToSplit = i

bestNewCents = centroidMat

bestClustAss = splitClustAss.copy()

lowestSSE = sseSplit + sseNotSplit

#该族被划分成两个子族后,其中一个子族的索引变为原族的索引，另一个子族的索引变为len(centList),然后存入centList

bestClustAss[np.nonzero(bestClustAss[:,0] == 1)[0],0] = len(centList)

bestClustAss[np.nonzero(bestClustAss[:,0] == 0)[0],0] = bestCentToSplit

centList[bestCentToSplit] = bestNewCents[0,:].tolist()

centList.append(bestNewCents[1,:].tolist())

self.clusterAssment[np.nonzero(self.clusterAssment[:,0] == bestCentToSplit)[0],:]= bestClustAss

self.labels = self.clusterAssment[:,0]

self.sse = sum(self.clusterAssment[:,1])

self.centroids = np.asarray(centList)

def predict(self,X):#根据聚类结果，预测新输入数据所属的族，返回预测标签

#类型检查

if not isinstance(X,np.ndarray):

try:

X = np.asarray(X)

except:

raise TypeError("numpy.ndarray required for X")

m = X.shape[0]#m代表样本数量

preds = np.empty((m,))

for i in range(m):#将每个样本点分配到离它最近的质心所属的族

minDist = np.inf

for j in range(self.n\_clusters):

distJI = self.\_distEclud(self.centroids[j,:],X[i,:])

if distJI < minDist:

minDist = distJI

preds[i] = j

return preds

#调用训练部分

# 二分聚类

km=biKMeans(n\_clusters)

cluster1=km.fit(array1)

y\_pred=km.labels

## 4.6 标签匹配和评估指标

### 4.6.1 KM算法、纯度计算代码

#纯度计算以及标签映射部分

from munkres import Munkres,print\_matrix

from sklearn.metrics import accuracy\_score

def best\_map(L1,L2):#映射

#L1是真实标签

#L2是聚类标签

Label1 = np.unique(L1) # 去除重复的元素，由小大大排列

nClass1 = len(Label1) # 标签的大小

Label2 = np.unique(L2)

nClass2 = len(Label2)

nClass = np.maximum(nClass1,nClass2)#选择最大值

G = np.zeros((nClass,nClass))#生成图G

#KM算法迭代部分：

for i in range(nClass1):

ind\_cla1 = L1 == Label1[i]

ind\_cla1 = ind\_cla1.astype(float)

for j in range(nClass2):

ind\_cla2 = L2 == Label2[j]

ind\_cla2 = ind\_cla2.astype(float)

G[i,j] = np.sum(ind\_cla2 \* ind\_cla1)

m = Munkres()

index = m.compute(-G.T)

index = np.array(index)

c = index[:,1]

newL2 = np.zeros(L2.shape)

for i in range(nClass2):

newL2[L2 == Label2[i]] = Label1[c[i]]#映射结果

return accuracy\_score(y\_true, newL2),newL2#返回纯度和映射结果

acc,y\_map=best\_map(y\_true,y\_pred)#调用该函数

### 4.6.2 其他指标评估代码

#评估代码

#已知标签评估，对预测标签和真实标签进行评估

#互信息分

info=metrics.mutual\_info\_score (y\_pred, y\_true)

#调整后的互信息分

mutual\_info=metrics.adjusted\_mutual\_info\_score (y\_pred, y\_true)

#标准化互信息分

normal\_info=metrics.normalized\_mutual\_info\_score (y\_pred, y\_true)

print("互信息分：{0}，调整后的互信息分：{1}，标准化：{2}".format(info,mutual\_info,normal\_info))

#兰德系数

lande=metrics.adjusted\_rand\_score(y\_pred, y\_true)

print('兰德系数：',lande)

#同质性、完整性以及其调和平均

a,b,c=metrics.homogeneity\_completeness\_v\_measure(y\_true, y\_pred)

print('同质性：{0}，完整性{1}，调和平均{2}'.format(a,b,c))

#未知标签评估，对簇本身进行评估

lunkuo=metrics.silhouette\_score(array1,y\_pred, metric='euclidean')#轮廓系数

print('轮廓系数：',lunkuo)

#标签映射结果和错误的划分结果存储以便后续分析：

okmap={}

rongmap={}

for i in range(len(y\_pred)):

if y\_true[i]==int(y\_map[i]):

ct+=1

okmap[y\_pred[i]]=y\_true[i]#获取预测标签所对应的真实标签

else:

# rongmap['t'+str(y\_true[i])+'->'+str(y\_map[i])]+=1

print("错误划分{0}->{1}".format(y\_true[i],y\_map[i]))

cf+=1

## 4.7 预测部分代码

预测数据——基于前面求好的质心，再计算到质心的值，根据距离最小原则，选择对应的类别，利用标签映射获得“预测的真实类别”：

with open(testpath,'rb')as f:

testset=pickle.load(f)#调用之前分好词的所有测试集

sampleoftextstest=testset['filecontent']#获取样本

print('测试集共{0}个'.format(len(sampleoftextstest)))#输出样本个数

# 测试集计算TF-IDF

import nltk

from nltk.text import TextCollection

corpus = TextCollection(sample)#语料库(之前的训练集)

sample\_test=[]#存储测试集所有特征向量

for j in sampleoftextstest:

singletext=j

words=nltk.word\_tokenize(singletext) #单词列表

tfidf\_words={}

#计算tfidf

for word in words:

idf=corpus.idf(word) #tf

tf=corpus.tf(word, words) #idf

tfidf=idf\*tf

tfidf\_words[word]=tfidf

#根据语料库词特征构建词向量

test\_tfvec=[]

# 对于每个已有特征词，如果文章不存在该词，其值设定为0，否则，计算文章在训练集语料库中的tf-idf值

for i in list(vec.get\_feature\_names()):

if i not in tfidf\_words.keys():

test\_tfvec.append(0.)#不存在设置为0

else:

test\_tfvec.append(tfidf\_words[i])

sample\_test.append(test\_tfvec)

sample\_test=np.array(sample\_test)

# 获取聚好的类的质心

centroid = cluster.cluster\_centers\_

result=[]

for i in sample\_test:

resulti=[]

for c in centroid:

#计算测试点到各个质心的距离

vecA=i

vecB=c

resulti.append(np.linalg.norm(vecA - vecB))

#根据最近距离选出所属簇的质心

result.append(resulti.index(min(resulti)))

y\_testpred=result#预测值

print(y\_testpred)

y\_testtrue=get\_kind(testset['filename']) #根据名字获取真实类别比对

countok=0

for i in range(len(y\_testpred)):

if okmap[y\_testpred[i]]==y\_testtrue[i]: #标签映射结果和真实结果比对得出准确率

countok+=1

print('测试集准确率：',countok/len(y\_testpred))

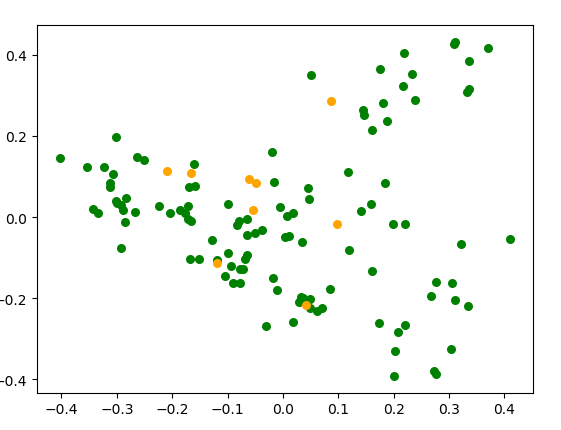
# 5 实验结果和分析

## 5.1 直观显示

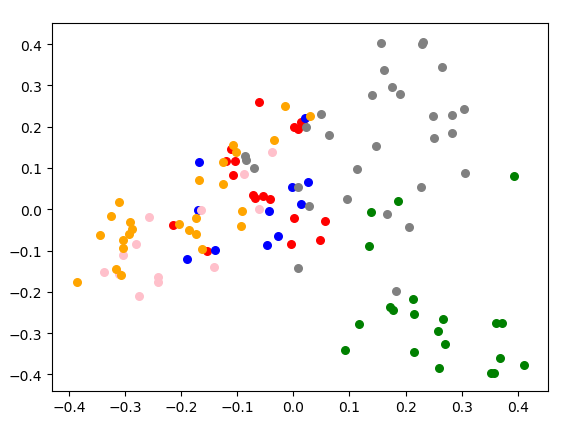
由于特征维度高，在高维数据图像展示的时候需要利用降维方法，这里采用PCA降维，降至二维展示。

这里给出各种K-Means的聚类效果图：（采用不同的颜色绘制六个簇）

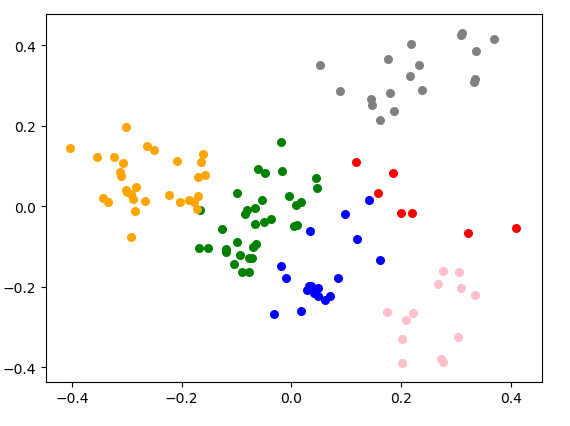
普通K-Means：



K-Means++：



二分K-Means：



注意，由于前两个K-Means是先聚类后降维显示，故某些情况下降维后的图像上不能展示出明显的簇结构。但依然可见各类别的是相对接近的。但是，上图中，二分K-Means是先降维后聚类的效果，因为经过多次测试发现，对于二分K-Means先降维有助于其预测准确率的提高。

## 5.2 指标评估

经多次测试，给定参数min\_df和max\_df分别为0.05和0.7时两种K-Means效果最好，依次比较几类评估指标：（取四位小数）

其中，簇个数均为6。普通K-Means和K-Means在直接迭代的情况下比先降维后迭代效果更好，而二分K-Means先降维后效果更佳，这里给出每一种K-Means的最优情况：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 聚类纯度（准确率） | 标准化互信息 | 兰德系数 | 轮廓系数 | 测试集分类成功率（测试准确率） |
| 普通K-Means | 0.5254 | 0.2210 | 0.2441 | 0.1439 | 0.4653 |
| K-Means++ | 0.7833 | 0.5885 | 0.4659 | 0.2427 | 0.7954 |
| 二分K-Means | 0.8322 | 0.3122 | 0.4444 | 0.4121 | 0.8363 |

可见，利用PCA降维后再使用二分K-Means的效果最好，普通K-Means效果较差。K-Means++的效果也不错。

上述表格中兰德系数均大于0，说明聚类都是成功的。

## 5.3 原因分析

①对于参数min\_df和max\_df分别为0.05和0.7时，聚类的准确率比其他参数更高，原因如下：

文档频率在0.05和0.7以外的词汇被筛选出去，即在120个文件中，同时出现在只有6个文件或者更少文件数量下的词汇可以被认为是非常不重要的词汇、噪声；在选择train时每一类都抽取了20个文件，可见文档频率低于0.05的词汇甚至在某一类中同时出现的概率最高只有0.3，无法表征某一类，并不适用于作特征词汇；

而文档频率在70%以上的词汇，出现概率过大，无法用于区分更细小的类别，也应当认为是临时的停用词，筛选出去。

②二分K-Means和K-Means++均在原始算法上做了改进，解决了K-Means本身易陷入局部最优、重启算法后效果不同以及运算速度慢的缺点。故相对地，聚类准确率和测试集准确率都高一些。

③很显然，由聚类的结果再去做分类会导致很大的错误率，三个算法分类成功率都并不高，考虑有以下原因：

* 样本本身问题；
* 由于TFIDF算法固有的缺点——严重依赖语料库，需要选取质量较高且和所处理文本相符的语料库进行训练；
* K-Means自身的缺陷：若簇中含有异常点，将导致均值偏离严重（即:对噪声和孤立点数据敏感）
* K-Means本身是随机选取质心的，一开始选取的不同会导致结果差异较大，所以在使用普通K-Means进行聚类以帮助分类时，效果并不稳定。

④此外，一开始我是挑选了每个类别中的20%作为测试集、剩下的作为训练集，但是庞大的特征向量空间不得不使用PCA主成分分析降维，否则效率非常低；应用降维后的效果要比之前好一些，但是聚类准确率和检测成功率仍然很低，最后找到问题是由于样本分布不均导致的。之后改成每一类取20个文件作为训练集，效果改善了很多。