山东大学
SHANDONG UNIVERSITY

实验报告

课　　程：	Data Mining
实验题目：	Homework 1
姓　　名：	陈昕
学　　号：	201814806
班　　级：	2018 级计算机学硕班
实验时间：	2018/10/8 – 2018/11/5

一、 实验目的

用 git 创建项目并上传到 GitHub 上，并学会使用 git 的基本命令。

理解课堂上学到的 Vector space model 和 KNN 分类方法，并在 20 Newsgroups 数据集上实践课堂上学到的文本处理、tf-idf、cosine similarity、反向索引及 KNN 分类等知识。

任务：

1. 预处理文本数据集，并且得到每个文本的 VSM 表示。
2. 实现 KNN 分类器，测试其在 20Newsgroups 上的效果。

二、 实验环境

硬件环境：

intel i5-7400 CPU @ 3.00 GHz
8.00 GB RAM

软件环境：

Windows 10 64 位 家庭版
python 3.6.6
Anaconda custom (64-bit)
Jupyter notebook 5.7.0
nltk 3.3.0

三、 实验过程

（我不能保证在报告描述中变量名与实际程序中完全一致。）

准备工作

从 https://git-scm.com/downloads 上可以下载到最新的 git 发行版。

在 GitHub 上创建账号，然后按 start a project 创建项目。

整个实验主程序的入口是 main.ipynb，可以通过 jupyter notebook 运行。

在任务正式开始之前，先从 http://qwone.com/~jason/20Newsgroups/ 上下载了 20news-18828.tar.gz 数据集，并用 7zip 将其解压。

任务一：预处理文本数据集，并且得到每个文本的 VSM 表示。

我在这个任务中使用 nltk(Natural Language Toolkit)来进行文本的预处理，将文本转化为词语集合，并自己编写了一个类 VectorSpaceModel 来把词语集合转换为 VSM 表示。

根据面向对象的单一职责原则，VectorSpaceModel 类没有文本预处理功能，文本预处理过程在 main.ipynb 中进行。

在 main.ipynb 中的文本预处理和转化为 VSM 的过程如下：

首先，用 os 模块获取 20Newsgroups 数据集的所有文件路径，filepaths。

将 filepaths 按照 8：2 的比例划分为训练集 training_set_fns 和测试集 test_set_fns（fn 是 file name 的缩写）。

然后，用文件读取函数 open()打开 training_set_fns 中的所有文档。**文档的编码方式为 latin1。**

Nltk 包中提供了 stopwords，tokenizer 和 stemmer。先用 RegexpTokenizer 从文档中取出所有单词，然后把所有词转换为小写并用 stemmer 取词干，最后过滤掉所有属于 stopwords 的停词。

对测试集中的所有文档进行上述处理，得到一个分词后的文件列表，命名为 tokenized_docs。这样就结束了文本预处理过程。

VSM 表示的各种功能是在 VectorSpaceModel 类里实现的。

在 main.ipynb 里，遍历 tokenized_docs，对每个文档用 VectorSpaceModel 包装，并将其中的词语加入计算 DF，并在遍历结束后计算 IDF。完成后即可计算各个 VSM 里每个词的 tf-idf 权重，再把向量转为单位向量。

我实现了 sub-linear 和 maximum 两种 tf normalization 方法，还可以选择直接使用 tf。这些选择定义在一个枚举类 TF_Scale 中。

在 main.ipynb 中，我选择了使用 maximum 方法来 normalize 词频。

VectorSpaceModel.py 文件中：

包括两个类 TF_Scale 和 VectorSpaceModel。

枚举类 TF_Scale 里定义了 TF 的 3 种 normalization 方式：RAW 代表不 normalization，SUB_LINEAR 表示使用$tf(t,d) = \begin{cases} 1+\log c(t,d), \ if\ c(t,d)>0 \\ 0, \ otherwise \end{cases}$，MAXIMUM 表示使用$tf(t,d) = \alpha + (1-\alpha)\frac{c(t,d)}{\max\limits_{t} c(t,d)}$。

类 VectorSpaceModel 的类属性包括：

- **rawDF**（整个词典和文档频率的字典）
- **_IDF**（值为 IDF 值的字典）
- **alpha**（进行 Maximum TF scaling 时的α）

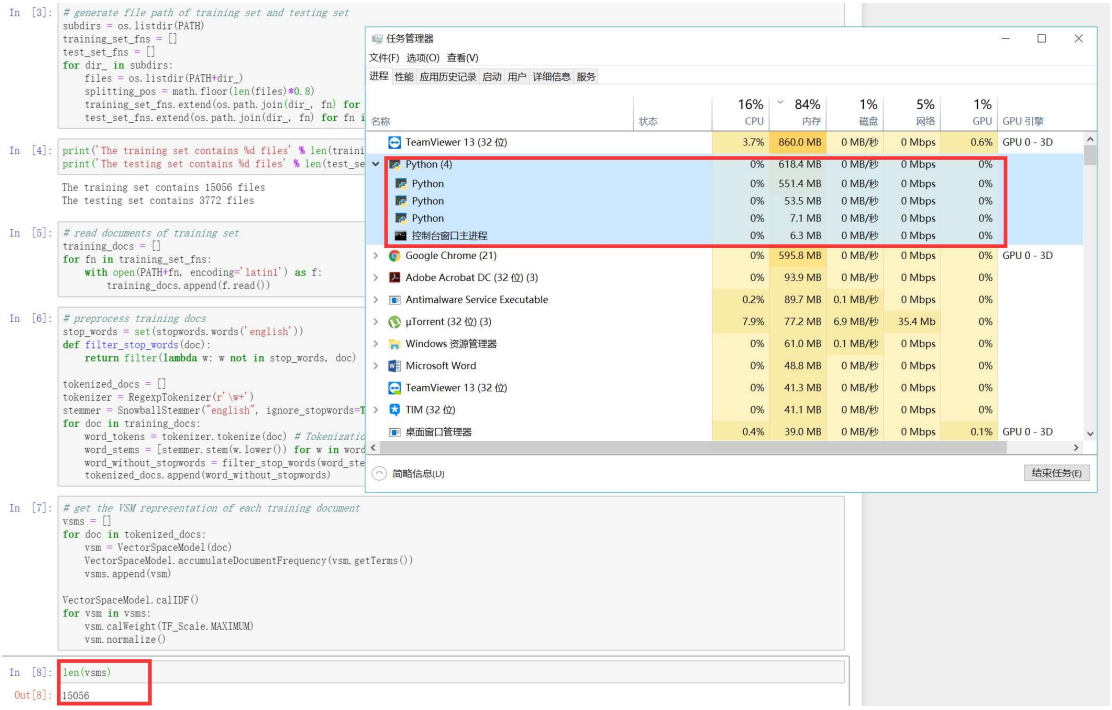
对象属性包括：

- **rawTF**（基于内置 Counter 的词频统计字典）
- **_maxCountInTF**（进行 Maximum TF scaling 时的最高词频）
- **vector**（存放存在于文档中的词以及它们的权重）。

类 VectorSpaceModel 的方法都比较简单，包括：

- calTF(根据 TF_Scale 所指定的方法计算 tf)
- calIDF（计算 IDF）
- accumulateDocumentFrequency（添加新文档信息到 rawDF）
- calWeight（计算 vector）
- normalize（把向量单位化）
- getTerms（返回文档中包含的所有词）
- getCorpus（返回全局的所有词语）
- dot 和 dotProduct（用两个标准化后的 VSM 进行点乘）。

我没有选择使用矩阵来记录各 VSM 表示，因为一个文档基本不可能拥有语料库中的所有词，用矩阵表示有极大可能得到稀疏的矩阵，不必要的 0 会占用过多的内存空间。



如图所示，在用 VectorSpaceModel 的对象列表来替代矩阵的情况下，尽管列表里包含 15056 个文档且仅使用 stopwords 过滤，内存消耗也只需要约 600MB，不会撑爆 8G 内存。

而用字典进行向量单位化和点乘也是节省时间与内存的，因为如果一个词的词频为 0，那么它不会影响这两个过程的结果，反而会花费不必要的时间。

任务二：实现 KNN 分类器，测试其在 20Newsgroups 上的效果。

我的 KNN 分类器实现在 KNNClassifier.py 中，测试代码依旧写在 main.ipynb 中。

在 main.ipynb 中的进行的分类过程如下：

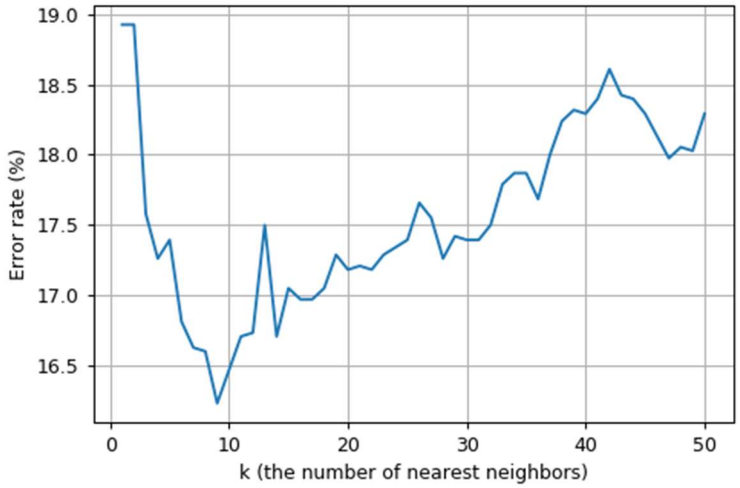
首先，读取测试集 test_set_fns，并转化为 VSM 表示。

接着构建 KNN 分类器。训练集的标签 labels 来自它们的文件夹名，可通过 os.path.dirname 得到。用训练集数据和 labels 就可以构建 KNNClassifier，而 train 函数中构建了反向索引。

然后，以从 1 开始的 50 个 k 作为参数，用分类器对测试集进行分类。

在分类完成后，对比分类器生成的 label 和测试集的实际 label（也就是文件夹名），计算错误率。

通过 numpy 包的方法，找到最佳 k 值，并用 matplotlib 画出错误率曲线，如下图所示：



结果显示在 k=9 时错误率最小，约 16.2%。之后虽然有些波动，但趋势是向上的，也就是说 k 并不是越大越好。

KNNClassifier.py 文件中：

该文件只包括 KNNClassifier 一个类。

类 KNNClassifier 中，没有类属性。

对象属性包括：

- **vsms**（训练数据集的 vsm 列表）
- **labels**（每个数据对应的标签）
- **invertedList**（作为反向索引的字典）。

对象方法包括：

- **buildInvertedList**（构建反向索引）
- **train**（内部调用 buildInvertedList 方法）
- **classify**（使用 inverted list 和 heap 来加速找到 k 近邻，用 Counter 求出 k 近邻中最多的 label,作为分类结果）

train 方法的存在是为了我想象中的 Classifier 接口，它应该有 train 和 classify 两个方法，所有 client 都应该只需要调用这两个方法，减小调用方的成本。

classify 方法是分类的主方法，要被调用多次，所以我使用反向索引和 heapq 模块提供的 nlargest 方法来进行加速。

四、 实验结论

在本次实验中，我动手实现了课堂上学到的文本预处理、vector space model 构建和 KNN 分类，感觉对这些知识点的理解更深入了。

在读取文件时，我发现 utf8 有些无法解析的字符，在网上查找后发现换用 latin1 编码就能解析全部文件了。这是因为基本上各种编码都兼容 ASCII 字符，而各种文本编辑器优先把只包含 ASCII 字符的文件是 utf8 编码的，但这样在碰到 latin1 字符时就会出现解析错误。

我还发现了 nltk 这个强大的自然语言处理工具，用它进行了分词、取词干、过滤停词的操作。它还有词性标记、解析、获取语义等功能，但我在本次实验中没有用到。

在实现 vector space model 时，我实现了 sub-linear 和 maximum 两种 tf normalization 方式，并用 dict 作为 vector，且实现了 vector 点乘。

在进行 knn 分类时，我对从 1-50 的 k 进行了测试，最高正确率约 84%，此时 k 取 9。错误率先下降再上升，出现了明显的转折点，所以存在最佳的 k 值。

山东大学
SHANDONG UNIVERSITY

实验报告

课　　程:	Data Mining
实验题目:	Homework 2
姓　　名:	陈昕
学　　号:	201814806
班　　级:	2018 级计算机学硕班
实验时间:	2018/11/6 – 2018/11/25

一、 实验目的

理解课堂上学习的朴素贝叶斯分类器（Naïve Bayes Classifier）的原理，包括贝叶斯公式、条件独立假设、不同模型的平滑技术及取对数等工程上的tricks。

任务：

3. 实现朴素贝叶斯分类器，测试其在 20Newsgroups 数据集上的效果。

二、 实验环境

硬件环境：

intel i5-7400 CPU @ 3.00 GHz
8.00 GB RAM

软件环境：

Windows 10 64 位 家庭版
python 3.6.6
Anaconda custom (64-bit)
Jupyter notebook 5.7.0
nltk 3.3.0
numpy 1.15.4
scikit-learn 0.20.0

三、 实验过程

（我不能保证在报告描述中变量名与实际程序中完全一致。）

准备工作

这一次实验中，并没有什么需要下载的东西。
我新建了 Homework2 文件夹，用于保存本实验中的所用到的程序文件等。
然后，将包含 20Newsgroups 数据集的文件夹复制到 Homework2 下。

任务一：实现朴素贝叶斯分类器，测试其在 20Newsgroups 数据集上的效果。

在这个任务中，我依然使用 nltk 进行文本预处理，并使用 scikit-learn 来划分训练集和测试集，并编写了一个 NaiveBayesClassifier 类来封装朴素贝叶斯分类器的实现。

在 main.ipynb 中的主程序过程如下：

首先，读取 20Newsgroups 数据集中所有文件的文件名 filepaths。

Scikit-learn 的 model_selection 包提供了一个 train_test_split 方法，可以在划分测试集和训练集的同时进行洗牌。我采用了该方法，把 filepaths 按 8：2 划分成了训练集和测试集，并读取了所有文件。

和 Homework1 一样，我用 nltk 进行了分词、取词干和去除停词的操作。

对于训练集，我将文件夹名作为 label，和预处理后的文档一起作为参数构建了一个 NaiveBayesClassifier，并进行训练。

对所有测试集中的数据，用分类器计算分类结果。

最后，对比分类器生成的 label 和测试集的实际 label（也就是文件夹名），计算错误率。

```
In [9]:
# create a naive Bayes classifier
labels = [os.path.dirname(fn) for fn in training_set_fns]
classifier = NaiveBayesClassifier(tokenized_docs, labels)

In [10]:
# training the classifier
classifier.train()

In [11]:
# classify the testing data for different k
results = []
for doc in whole_test_tokenized_docs:
    results.append(classifier.classify(doc))

In [12]:
error_rate = 0
sum_ = 0
for i, label in enumerate(results):
    if label != os.path.dirname(test_set_fns[i]):
        sum_ += 1

error_rate= sum_/N
print('The accuracy is: %f' % (1-error_rate))

The accuracy is: 0.831121
```

结果显示，我实现的朴素贝叶斯分类器对于所有数据集的正确率大约是 83%。这个结果实际上略低于我在 KNN 分类器中得到的最高正确率（84%），但是 KNN 分类器大约需要一个小时来完成整个分类过程，但朴素贝叶斯分类器的分类过程要快得多，只需要大约一分钟。

NBC.py 文件中：

该文件只包括 NaiveBayesClassifier 一个类。

类 NaiveBayesClassifier 中，没有类属性。

对象属性包括：

- training_data（训练数据集列表）
- labels（每个数据对应的标签）
- label_missing_word_prob（每个类对应的不属于该类的词的概率，用于平滑技术）。
- label_word_probability（每个类对应的属于该类的词的概率）。

- label_class_prob（训练集中，各类文档数占总数的比例）。

对象方法包括：

- train（计算对象属性里提到的各种概率，并对它们取对数）
- classify（统计文档中词的次数，求文档属于各类的概率，然后选出使得概率最大的那个类）

可以看出，我在这个朴素贝叶斯分类器的实现中采用了多项式模型。

在 train 阶段，我使用 python 内置的 Counter 进行统计词语在每一类中出现的次数总和，并统计词表长度。然后，用平滑技术计算词语概率，并用 math.log 对所有概率取对数。

在 classify 阶段，我首先把分词后的文档转化为了一个 Counter，并遍历所有 label，用相应的取对数后的类概率加上所有集合中的词的出现概率乘以出现次数，求出概率最大的类标签作为结果。

四、 实验结论

在本次实验中，我动手实现了朴素贝叶斯分类器，其中用到了课堂上学到的贝叶斯公式、条件独立假设、不同模型的平滑技术及取对数等工程上的 tricks，感觉对这些知识点的理解更深入了。

我对上述知识点的理解是，朴素贝叶斯分类器基于贝叶斯公式，将先验知识（训练集中的文档在类中的概率和类在总体中的概率）转换为后验知识（新文档属于哪个类）。它还引入了条件独立假设，也就是假设所有词的出现概率是相互独立的，所以可以把文档在类中的概率转换成词语在类中的概率中的累乘。平滑技术用于处理词不在类中的情况，包括伯努利模型、多项式模型和混合模型；而取对数技术把累乘转化成了累加，解决了浮点数下溢问题并提高了计算速度。

在实验过程中，我还犯过好几个错误，也学到了不少东西。

最初，我是使用迭代器作为 training_data 的来源，这样重复调用 train 方法就会导致统计结果为空，结果正确率只有 20%多。在我用 list 展开迭代器后，重复运行就正常了。

其次，我还忘记了计算类概率时也要取对数，结果正确率不到 60%。

我是通过 debug 工具定位这些问题的。使用 import pdb; pdb.set_trace() 可以设置断点，直接在 jupyter notebook 中检查中间变量的结果。

这次我还采用了 scikit-learn 中的 train_test_split 方法来随机划分训练集和测试集，能够增强结果的健壮性和可信度。

最终结果显示朴素贝叶斯分类的正确率大概 83%，稍低于 KNN 分类器的最好情况（k=9，正确率 84%）；但是运行速度比 KNN 分类器快得多，训练和分类加起来只要大概一分钟，而 KNN 分类器需要 1 个多小时。

山东大学
SHANDONG UNIVERSITY

实验报告

课　　程：	Data Mining
实验题目：	Homework 3
姓　　名：	陈昕
学　　号：	201814806
班　　级：	2018 级计算机学硕班
实验时间：	2018/11/25 – 2018/12/24

一、 实验目的

测试 sklearn 中以下聚类算法在 tweets 数据集上的聚类效果，并使用 NMI(Normalized Mutual Information)作为评价指标。

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n_samples`, medium `n_clusters` with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters, linkage type, distance	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers

通过该实验，掌握常用聚类算法的思想及调用方式。

二、 实验环境

硬件环境：

intel i5-7400 CPU @ 3.00 GHz
8.00 GB RAM

软件环境：

Windows 10 64 位 家庭版
python 3.6.6
Anaconda custom (64-bit)
Jupyter notebook 5.7.0
numpy 1.15.4
scikit-learn 0.20.0

三、 实验过程

准备工作

首先，读取 Tweets.txt，其中每一行是是一个 json，可以用 json.loads 读取。其中，text 指向文本，也就是我们要用到的数据，而 cluster 表示每条数据属于哪个类。

In [4]:
```
# construct tf-idf matrix
corpus = [d['text'] for d in data]
vectorizer = TfidfVectorizer(min_df=2, sublinear_tf=True)
X = vectorizer.fit_transform(corpus).todense()
tfidf_matrix = np.array(X)
```

因为文本不能直接用于聚类，所以我使用了 sklearn 中的 TfidfVectorizer 将文本数据转化为包含 tfidf 值的向量。使用 min_df=2 来保证只有文档频率大于 1 的词才会被加入字典，加快计算，sublinear_tf=True 来让 tf 被标准化。

因为这样生成的矩阵是稀疏的(scipy sparse matrix)，所以要用 todense()转化它，以便聚类方法的计算。最后用 np.array 来把矩阵转为 array-like 的。

In [5]:
```
# get labels of each tweet
labels = [d['cluster'] for d in data]
```

labels 是每条数据的实际标签，用于和聚类算法预测出的标签对比。

sklearn.metrics. **normalized_mutual_info_score** (*labels_true, labels_pred, average_method='warn'*) [source]

NMI(Normalized Mutual Information, 规范互信息)可以用于比较两个聚类结果的相似性，范围为 0.0（无互信息）-1.0（完美匹配）。我使用的 average_method 是‘arithmetic’，因为使用‘warn’会出现警告，而文档中介绍 0.22 版中将使用‘arithmetic’作为默认选项。

这种方法的主要缺点就是要知道数据真正属于哪个类，而很多场景下聚类就是因为不知道类信息。但是这一点对我们的数据来说刚刚好。

K-Means

In [6]:
```
# K-Means clustering test
k_means = KMeans(n_clusters=max(labels), n_jobs=-1).fit(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, k_means.labels_, average_method='arithmetic')
print('The NMI score of K-Means is:', nmi)
```

```
The NMI score of K-Means is: 0.8083287921393979
```

K-Means 聚类方法是很简单又高效的聚类方法，目标是最小化各点到类中心的距离的平方和。

但 sklearn 的实现没有办法指定使用的距离函数，所以只能用欧氏距离计算。尽管如此，最后算出的 NMI 分数为 0.8083287921393979，效果还是不错的。n_clusters 参数指定了要聚成几类，相当于 K；n_jobs 指定要使用几个核计算，-1 表示使用所有可用核。

Affinity propagation

In [7]:
```
# Affinity propagation clustering test
ap = AffinityPropagation().fit(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, ap.labels_, average_method='arithmetic')
print('The NMI score of Affinity propagation is:', nmi)
```

```
The NMI score of Affinity propagation is: 0.7733356331577484
```

Affinity propagation 通过在样本对之间发送数据创建集群，直到收敛。我没有修改默认参数，最后算出的 NMI 分数为 0.7733356331577484，效果

还行。

Mean Shift

```
In [8]: # MeanShift clustering test
        ms = MeanShift(bandwidth=0.44, bin_seeding=True, min_bin_freq=2, cluster_all=True, n_jobs=-1).fit(tfidf_matrix)
        nmi = normalized_mutual_info_score(labels, ms.labels_, average_method='arithmetic')
        print('The NMI score of MeanShift is:', nmi)
        print('Its shape is:', np.unique(ms.labels_).shape)

        The NMI score of MeanShift is: 0.6901183198703617
        Its shape is: (85,)
```

Mean Shift 聚类旨在发现样本的平滑密度中的“斑点”。它和 KMeans 一样是基于中心的，先更新中心候选为每个给定区域的点的平均值，然后在后处理阶段过滤候选来消除近似重复，形成最后的质心集。

为了选出质心，最重要的参数是 bandwidth，如果使用默认 bandwidth 或 bandwidth=1 都会导致最后结果只包含 1 个类。bandwidth=0.44 算是一个相对较好的 bandwidth，但 NMI 分数也只有 0.6901183198703617，属于较低的。

bin_seeding=True， min_bin_freq=2， n_jobs=-1 的设置是为了加速计算，而 cluster_all=True 的设置是为了让所有点都属于一个类。

Spectral Clustering

```
In [9]: # SpectralClustering test
        sc = SpectralClustering(n_clusters=max(labels)).fit(tfidf_matrix)
        nmi = normalized_mutual_info_score(labels, sc.labels_, average_method='arithmetic')
        print('The NMI score of SpectralClustering is:', nmi)

        The NMI score of SpectralClustering is: 0.698345651992622
```

谱聚类方法在样本之间进行低维度嵌入亲和度矩阵，然后在低维空间中进行 KMeans。根据文档描述，当各聚类结构高度非凸时，谱聚类十分有用。但对我们的数据，最后算出的 NMI 分数为 0.698345651992622，效果也不太好。

n_clusters 参数指定了要聚成几类。

Ward hierarchical clustering

```
In [10]: # Ward hierarchical clustering test
         whc = AgglomerativeClustering(n_clusters=max(labels), linkage='ward').fit(tfidf_matrix)
         nmi = normalized_mutual_info_score(labels, whc.labels_, average_method='arithmetic')
         print('The NMI score of Ward hierarchical clustering is:', nmi)

         The NMI score of Ward hierarchical clustering is: 0.7864984641747731
```

Ward hierarchical clustering 是层次聚类的一种，Ward 表示最小化所有聚类内的平方差的总和。这个优化目标与 K-Means 的目标很相似，但是计算方法是 agglomerative hierarchical approach。

最后算出的 NMI 分数为 0.7864984641747731，还算不错。

n_clusters 参数指定了要聚成几类，linkage='ward' 指定了链接方法。文档中提到了当 linkage 为 ward 时，只能使用 Euclidean 方式计算亲密性。

Agglomerative clustering

In [11]:
```
# AgglomerativeClustering test
ac = AgglomerativeClustering(n_clusters=max(labels), affinity='cosine', linkage='single').fit(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, ac.labels_, average_method='arithmetic')
print('The NMI score of Agglomerative Clustering with single linkage is:', nmi)
ac = AgglomerativeClustering(n_clusters=max(labels), affinity='cosine', linkage='average').fit(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, ac.labels_, average_method='arithmetic')
print('The NMI score of Agglomerative Clustering with average linkage is:', nmi)
```

```
The NMI score of Agglomerative Clustering with single linkage is: 0.22496764238754718
The NMI score of Agglomerative Clustering with average linkage is: 0.9003064131754916
```

凝聚聚类是一种通用的聚类算法系列，它通过连续合并或拆分嵌套聚类来构建嵌套聚类，上课时也重点讲了。这部分实验中我使用了其他 linkage 方式。

n_clusters 参数指定了要聚成几类，affinity='cosine'指定了用 cos 相似度来计算亲密性，linkage 选用了 single（最小化两类间最近点对的距离）和 average（最小化两类间所有点对的平均距离）方式。

可以看到，使用 single 方式计算出的 NMI 分数很低，大约为 0.23；而使用 average 方式计算出的分数就很高，大约为 0.9，这也是我计算出的最高 NMI 分数。

DBSCAN

In [12]:
```
# DBSCAN test
dbscan = DBSCAN(eps=0.1, n_jobs=-1).fit(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, dbscan.labels_, average_method='arithmetic')
print('The NMI score of DBSCAN with eps=0.1 is:', nmi)
print('Its shape is:', np.unique(dbscan.labels_).shape)
dbscan = DBSCAN(eps=1, n_jobs=-1).fit(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, dbscan.labels_, average_method='arithmetic')
print('The NMI score of DBSCAN with eps=1 is:', nmi)
print('Its shape is:', np.unique(dbscan.labels_).shape)
dbscan = DBSCAN(eps=2, n_jobs=-1).fit(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, dbscan.labels_, average_method='arithmetic')
print('The NMI score of DBSCAN with eps=2 is:', nmi)
print('Its shape is:', np.unique(dbscan.labels_).shape)
```

```
The NMI score of DBSCAN with eps=0.1 is: 0.027537116536474605
Its shape is: (5,)
The NMI score of DBSCAN with eps=1 is: 0.568925646557483
Its shape is: (64,)
The NMI score of DBSCAN with eps=2 is: -8.557811214631876e-17
Its shape is: (1,)
```

DBSCAN 算法将 cluster 视为由低密度区域分隔的高密度区域，高密度区域的连接半径由 eps 定义。

可以看出，当 eps=0.1 时和 eps=2 时，分类效果都非常差，NMI 很低。

而 eps=1 时，NMI 分数相对较高，为 0.568925646557483，但相对于其他算法来说 DBSCAN 的聚类效果还是属于最差的一档。

Birch

In [13]:
```
# Birch test
brc = Birch(n_clusters=max(labels)).fit(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, brc.labels_, average_method='arithmetic')
print('The NMI score of Birch is:', nmi)
```

```
The NMI score of Birch is: 0.8050399281197088
```

Birch 为给定数据构建一个称为特征树（CFT， Characteristic Feature Tree）的树。数据基本上被有损压缩为一组 CF 节点。CF 节点有许多称为特征子集（CF Subclusters）的子集群，位于非终端 CF 节点中的这些 CF 子集可以将 CF 节点作为子节点。

这种方法最后算出的 NMI 分数为 0.8050399281197088，效果比较好。

n_clusters 参数指定了要聚成几类。

Gaussian Mixture

In [14]:
```
# Gaussian mixtures test
gm_labels = GaussianMixture(n_components=max(labels), covariance_type='tied').fit_predict(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, gm_labels, average_method='arithmetic')
print('The NMI score of Gaussian mixtures is:', nmi)
```

```
The NMI score of Gaussian mixtures is: 0.7987454617917258
```

Gaussian Mixture 是一个概率模型，实际上不属于 cluster 而是属于 mixture 包。它假设所有数据点是从具有未知参数的有限数量的高斯分布的混合生成的，也就是每个点有一定概率属于 a 类的同时属于 b 类。

这种方法最后算出的 NMI 分数为 0.7987454617917258，也属于效果较好的。

n_components 参数指定了里面包含几个 component，每个 component 为一个高斯核，相当于聚类算法中的 n_clusters。

当不指定 covariance_type 是，默认是对每个 component 都需要计算协方差矩阵，这会造成内存不足的错误。

```
# Gaussian mixtures test
gm_labels = GaussianMixture(n_components=max(labels)).fit_predict(tfidf_matrix)
nmi = normalized_mutual_info_score(labels, gm_labels, average_method='arithmetic')
print('The NMI score of Gaussian mixtures is:', nmi)
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-77-37d39106b86d> in <module>
      1 # Gaussian mixtures test
----> 2 gm_labels = GaussianMixture(n_components=max(labels)).fit_predict(tfidf_matrix)
      3 nmi = normalized_mutual_info_score(labels, ac.labels_, average_method='arithmetic')
      4 print('The NMI score of Gaussian mixtures is:', nmi)

~\Anaconda3\lib\site-packages\sklearn\mixture\base.py in fit_predict(self, X, y)
    243
    244                 log_prob_norm, log_resp = self._e_step(X)
--> 245                 self._m_step(X, log_resp)
    246                 lower_bound = self._compute_lower_bound(
    247                     log_resp, log_prob_norm)

~\Anaconda3\lib\site-packages\sklearn\mixture\gaussian_mixture.py in _m_step(self, X, log_resp)
    673         self.weights_, self.means_, self.covariances_ = (
    674             _estimate_gaussian_parameters(X, np.exp(log_resp), self.reg_covar,
--> 675                                           self.covariance_type))
    676         self.weights_ /= n_samples
    677         self.precisions_cholesky_ = _compute_precision_cholesky(

~\Anaconda3\lib\site-packages\sklearn\mixture\gaussian_mixture.py in _estimate_gaussian_parameters(X, resp, reg_covar, covariance_type)
    283                    "diag": _estimate_gaussian_covariances_diag,
    284                    "spherical": _estimate_gaussian_covariances_spherical
--> 285                    }[covariance_type](resp, X, nk, means, reg_covar)
    286     return nk, means, covariances
    287

~\Anaconda3\lib\site-packages\sklearn\mixture\gaussian_mixture.py in _estimate_gaussian_covariances_full(resp, X, nk, means, reg_covar)
    162     """
    163     n_components, n_features = means.shape
--> 164     covariances = np.empty((n_components, n_features, n_features))
    165     for k in range(n_components):
    166         diff = X - means[k]

MemoryError:
```

而 covariance_type='tied'让所有 component 共享协方差矩阵，这样就不会出现 Memory Error 了。

四、 实验结论

在本次实验中，我实验了 sklearn 中的多种聚类算法，并了解了 NMI 这一评价聚类算法效果的指标。

在我使用的这些方法中，效果最好的还是使用 linkage=‘average’，affinity=‘cosine’的凝聚聚类算法，最差的则是使用 linkage=‘single’的凝聚聚类算法和 DBSCAN。如果参数调节不好，那么 MeanShift 和 DBSCAN 都会产生很差的结果。

简单来说，在知道类标签数量的情况下，KMeans、Birch、Gaussian Mixture 和使用 linkage=‘average’的凝聚聚类算法可以达到很好的效果。如果不知道类标签数量，Affinity propagation 算法的效果相对比较好。

这次实验给我的感觉就是现在发现的聚类算法大多都需要控制参数，典型的比如 K-Means 的 k，而这些控制参数的调整其实不是很直观，如果我是有数据而不加以分析，是很难选出好的聚类算法，达到很好的聚类效果的。

所以，还是要从数据的特征入手，分析数据与聚类算法的相性。比如这次实验中使用的知道真实类标签的文本数据，就最适合有 cos 相似度计算的凝聚聚类方法，而且聚类间的链接方式使用 average 就比使用 single 这种找最小值的方式更好。而 DBSCAN 是一个效果不好的典型例子，它的假设是 cluster 是由低密度区域分隔的高密度区域，从直觉上来讲，文本数据在高维空间里很难是高密度的，所以无论怎么调节参数都不能获得很好的聚类效果。

通过该实验，我了解到了很多常用聚类算法的思想及调用方式，对聚类问题有了更深刻的认识，收获很大。