实验一　VSM

本实验先通过 tokenization, stemming,去 stopwords 等步骤对文档集 **20 Newsgroups dataset** 进行了预处理，然后建立了该文档集的词袋，并对其中每个文档建立了向量空间模型，最后用 TF-IDF 对其进行了改进。

1. 读取文档

读取文档时需要用到 python 自带的 os 模块遍历文档集中的每一个文档，以二进制方式打开，借助 chardet 模块的 detect 函数检测每个文档的编码方式，对一个文档中的每一行按其编码方式解码，并 join 为一个字符串。最终得到一个包含所有文档的列表，其中每个元素是由一个文档构成的字符串。

```
# 二进制方式读取，获取字节数据，检测类型
def get_encoding(file):
    with open(file, 'rb') as f:
        return chardet.detect(f.read())['encoding']
```

2. 预处理

建立一个文档预处理的函数 filter，借助 nltk 模块的 tokenization，stem，去 stopwords 等功能对文档进行预处理，由于还有一些没用的标点符号、连字符没被处理，自建了一个包含各种标点符号的列表，对出现在其中的符号进行删除，对连字符用空格进行了替换，另外将大写统一为了小写。

```
# 去标点
punctuation = [',', '<', '>', '.', "'s", '`', '~', '!', '@', '#', '$', '%', '^',
               '&', '*', '(', ')', '_', '', '=', '+', '[', ']', '{', '}', '\\', '|',
               ':', ';', "\'", '/', '?',"\"","“","’","``"]
filtered2 = [i for i in filtered1 if i not in punctuation]
```

3. 建立词袋并对每个文档统计词频

导入 scikit-learn 的 CountVectorizer，用 fit（）函数建立预处理后的文档集的词袋，并用 transform 函数对每个文档编码词频信息，得到一个所包含单词的稀疏向量。

```
# 对每个文档统计词频
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
# 建立词典
vectorizer.fit(sumData)
print(vectorizer.vocabulary_)

lenth = len(sumData)
for i in range(lenth):
    vector = vectorizer.transform([sumData[i]])
```

4. 用 TF-IDF 改进

借助 scikit-learn 的 TfidfVectorizer 建立文档集的词汇表并计算 idf 得分，再用 transform 函数对每个文档编码基于 TF-IDF 的词频得分。

```
# 对每个文档用 tf-idf 统计词频
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer1 = TfidfVectorizer()
# 建立词汇表
vectorizer1.fit(sumData)

lenth = len(sumData)
for i in range(lenth):
    vector1 = vectorizer1.transform([sumData[i]])
```

通过这次实验，我加深了对向量空间模型的理解，熟悉了 python 的编程语言。

实验二　朴素贝叶斯分类器

在实验一中，对文档集 20 Newsgroups dataset 建立了向量空间模型，本实验在实验一的基础上用朴素贝叶斯分类器实现了对文档的自动分类。

1.数据划分。为避免实验一中文件读取上的麻烦，直接从 sklearn.datasets 上获取 20newsgroups 文档集，并用 sklearn.model_selection 中的 train_test_split 函数实现对训练集和测试集的划分，这里训练集设为 80%，测试集设为 20%。

```
#训练数据与测试数据的划分
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split

newsgroups = fetch_20newsgroups()
X_train,X_test,y_train,y_test=
train_test_split(newsgroups.data,newsgroups.target,test_size=0.2,random_state=1)
```

其中 random_state 表示随机状态，这里设为 1，表示划分训练集和测试集的第一种随机状态。

2.Tfidf 向量化处理。分别对所划分的训练集和测试集的文档建立 Tfidf 权重的向量空间模型，同实验一。

```
#Tfidf 向量化处理
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(stop_words='english', lowercase=True)
#以 X_train 建立字典并向量化
train_vector = vectorizer.fit_transform(X_train)
test_vector = vectorizer.transform(X_test)
```

3. 训练并预测。用 sklearn.naive_bayes 中的 MultinomialNB 函数实现多项式模型的分类算法，对测试数据中的 0 频次项，采用拉普拉斯平滑，平滑因子 alpha 设置为 0.01，预算数据类别先验设置为 false，训练完成后直接对上一步得到的测试数据 Tfidf 向量化后的向量 test_vector 进行预测。预测结果为以测试集文档数目为长度的列表，每个元素为对每个文档所分的类别对应的索引。

```
#多项式模型分类

from sklearn.naive_bayes import MultinomialNB
#训练 拉普拉斯平滑参数 alpha 设为 0.01
mnb_clf = MultinomialNB(alpha=0.01, fit_prior=False)
mnb_clf.fit(train_vector, y_train)

# 预测
pred = mnb_clf.predict(test_vector)
print(pred)
```

4. 评分。采用 Micro-F1 进行打分，其中 precision 和 recall 分别为总体的精确率和召回率。

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

用 metrics.classification_report(y_test, pred)命令返回详细的打分结果如下：

document	precision	recall	f1-score	support
0	0.88	0.90	0.89	97
1	0.82	0.85	0.83	114
2	0.90	0.87	0.88	112
3	0.82	0.84	0.83	127
4	0.87	0.93	0.90	112
5	0.94	0.93	0.93	115
6	0.88	0.79	0.83	124
7	0.91	0.89	0.90	108
8	0.96	0.96	0.96	99
9	0.95	0.97	0.96	113
10	0.96	0.96	0.96	108
11	0.95	0.99	0.97	120
12	0.89	0.87	0.88	119
13	0.97	0.98	0.98	119

14	0.94	0.96	0.95	118
15	0.90	0.95	0.92	128
16	0.92	0.98	0.95	111
17	0.97	0.97	0.97	123
18	0.94	0.89	0.92	110
19	0.86	0.73	0.79	86
avg / total	0.91	0.91	0.91	2263

最终 F1 得分 0.91，说明分类效果较好。

通过这次实验，我对朴素贝叶斯分类器实现步骤及预测结果的评价方法有了更深入的了解。

实验三　倒排索引和布尔检索模型

1. 在 tweets 数据集上构建 inverted index

- 读取 tweets 数据集，将每个 tweets 信息作为列表中的一个字符串元素

```
tweets = []
path = 'C:\\Users\\hp\\Desktop\\IR 实验\\tweets.txt'
with open(path, 'rb') as file:
    lines = file.readlines()
    for line in lines:
        line = line.decode("ascii")
        tweets.append(line)
```

- 提取每个 tweets 的正文（第九个引号到第十个引号之间的部分）

```
def substring(str1):
    count = 0
    str = ""
    for s in str1:
        if s == '"':
            count = count + 1
        elif count == 9:
            str = str + s
    return str


tweets_text = []    #tweets_text 中的元素为原每个 tweet 的正文
for tweet in tweets:
    tweets_text.append(substring(tweet))
```

- 对每个 tweets 正文进行同实验一中的 tokenization，stemming，去 stopwords 等预处理
- 对上一步 tweets 文档中预处理后的单词建立字典，每个单词的 value 为其所在的

所有 tweets 的标签构成的集合，tweets 标签为之前 tweets 列表的索引值

```
dict1 = {}
label = 0
for pretext in textword:
    for word in pretext:
        if word not in dict1:
            dict1[word] = set()
        dict1[word].add(label)
    label = label + 1
```

2. 实现 Boolean Retrieval Model

- 这里使用集合间的运算实现检索。在检索开始前，创建 answer_set 集合用以盛放满足检索条件的 tweets 标签，在输出时，将 answer_set 转变为 list 并将元素排序后输出。

```
def print_answer(answer_set):                          #Output search results
    empty_set = set()
    if empty_set == answer_set:
        print('None')
    else:
        answer_list = list(answer_set)
        answer_list.sort()
        print(', '.join(map(str, answer_list)))
```

- 对于 OR 检索，需要输入以”OR:”开头的字符串，检索内容从第 3 个字符开始读取，在对 query 字符串进行相同预处理后，对于 query 中出现的每个单词，求它在字典中的 value 与 answer_set 的并集。

```
if 'OR:' in query:
    query_word = filter(query[3:])  #注意 query 也需要进行预处理
    print(query_word)
    for word in query_word:
        if (word != '')and(word in dict1):
            answer_set = dict1[word] | answer_set
```

- 对于 AND 检索，answer_set 需要初始化为含有所有文档的标签，query 从第 4 个字符开始读取，经过同样预处理后，对于 query 中出现的每个单词，求它在字典中的 value 与 answer_set 的交集。对于 NOT 检索，过程类似，最后求差集即可。

```
if 'AND:' in query:
    query_word = filter(query[4:])
    print(query_word)
    if query_word != []:      #第一次要跟所有文档标签做与操作，因此这里集合内容需初始化
        for i in range(30548):
            answer_set.add(i)
        for word in query_word:
            if word != '':
                if word in dict1:
                    answer_set = dict1[word] & answer_set
                else:
                    answer_set = set()
    else:
        print('None')
if 'NOT:' in query:
    query_word = filter(query[4:])
    print(query_word)
```

```
    for i in range(30548):
        answer_set.add(i)
    for word in query_word:
        if word != '':
            if word in dict1:
                print(list(dict1[word]))  #输出含有该单词的文档
                for i in dict1[word]:
                    answer_set.discard(i)
```

3. 测试

- OR:

  INPUT: OR:House Improve

  OUTPUT:

  ['hous', 'improv']

  0, 2, 100, 239, 291, 327, 522, 523, 537, 779, 803, 1018, 1069, 1316, ……

- AND:

  INPUT: AND:Ron Weasley birthday

  OUTPUT:

  ['ron', 'weasley', 'birthday']

  10314, 16613, 16745, 16787, 16792, 16794, 16809, 16817, 16820, ……

- NOT:

  INPUT: NOT:recall Skill says

  OUTPUT：

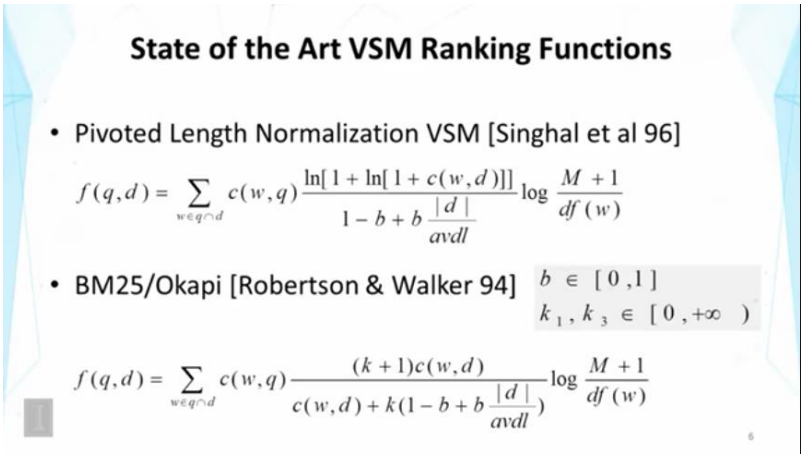
  ['recal', 'skill', 'say']

  3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25,……

经小范围验证，输出结果均正确。

实验四 Pivoted Length Normalization VSM and BM25

本次实验是在实验三布尔检索的基础上进行改进，对文档集中每个出现检索词的文档计算相似度评分，并将 doc_id 按其相似度从大到小的顺序输出。两种相似度评分公式如下：



1. 建立 inverted index

读取 tweets 数据集、截取正文、分词等预处理等步骤与实验三一致，在建立 inverted index 时，由于在计算 query 与文档相似度时需要用到 TF 与 DF, 因此建立倒排索引时仍然使用实验三中将字典的 value 设为集合的方法不再合适，这里字典中每个 word 的 value 设为一个列表，它的元素为包含此 word 的每个 doc_id 及其 c(w,d) 组成的列表。

```
#建立 inverted index
dict1 = {}
label = 0
for word_list in textword:
    for word in word_list:
        if word not in dict1:
            dict1[word] = []
            dict1[word].append([label, word_list.count(word)])
        elif label != dict1[word][-1][0]:
            dict1[word].append([label, word_list.count(word)])
    label = label + 1
```

2. 检索、打分

采用与实验三类似的思路，以两种方法的缩写“PLNVSM”，“BM25”作为输入的标识符，采取相应的公式进行计算。对 query 进行预处理，得到包含每个检索 word 的列表，对字典中 word 对应的每个文档进行打分（参数 b = 0.5，k = 2），将列表 score_list 作为 query 的打分结果，其每个元素为 doc_id 及其分数。

```
#检索、打分
while True:
    print("begin")
    query = input()
    if query == '':
        break
    if 'PLNVSM:' in query:
        query_word = filter(query[7:])  # 注意 query 也需要进行预处理
```

```
    if 'BM25:' in query:
        query_word = filter(query[5:])
    print('query 预处理: ', query_word)
    score_list = []
    for word in query_word:
        for d in dict1[word]:
            if 'PLNVSM:' in query:
                sim = query_word.count(word) * log(1 + log(1+d[1], math.e), math.e) / \
                      (1 - b + b * len(textword[d[0]])/avdl) * log((M+1)/len(dict1[word]), 2)
            if 'BM25:' in query:
                sim = query_word.count(word) * (k+1)*d[1]/(d[1]+k*(1-b+b*len(textword[d[0]])/avdl)) \
                      * log((M+1)/len(dict1[word]), 2)
            judge = True
            for i in score_list:
                if i[0] == d[0]:
                    judge = False
                    i[1] = i[1] + sim
            if judge:
                score_list.append([d[0], sim])
```

3. 排序、输出

按 score_list 的每个元素的打分结果高低顺序输出对应的 doc_id，用 Ron Weasley Birthday 进行测试，两种方法的结果如下：

```
D:\Anaconda3\python.exe C:/Users/hp/PycharmProjects/IR/IR.4.py
avdl =  11.757299986905853
begin
PLNVSM:Ron Weasley Birthday
query预处理:  ['ron', 'weasley', 'birthday']
文档评分:  [20.460633892762573, 20.460633892762573, 20.460633892762573, 20.460633892762573, 20.460633892762573, 19.565426864253077, 19.239635646699636, 19
对应doc_id:  [16745, 16787, 16938, 17011, 17023, 16745, 16787, 16938, 17011, 17023, 16745, 16787, 16938, 17011, 17023, 16745, 16787, 16938, 17011, 17023,
enter a doc_id to browse:
-1
begin
BM25:Ron Weasley Birthday
query预处理:  ['ron', 'weasley', 'birthday']
文档评分:  [35.25533328744369, 34.62665859906343, 33.59534559678648, 33.37775577756306, 33.37775577756306, 33.37775577756306, 33.37775577756306, 33.377755
对应doc_id:  [16792, 17069, 17022, 16745, 16787, 16938, 17011, 17023, 16745, 16787, 16938, 17011, 17023, 16745, 16787, 16938, 17011, 17023, 16745, 16787,
enter a doc_id to browse:
```

两种方法排序结果差别不大，互相验证，较为合理。

4. 浏览得到的文档

为了查看并验证检索得到的文档排序是否合理，可以对文档进行查询。

```
#文档查询
while True:
    print('enter a doc_id to browse:')
    doc_id = int(input())
    if doc_id == -1:
        break
    print(tweets_text[doc_id])
```

在搜索 Ron Weasley Birthday 完成后，查询排名靠前的 doc 的内容，发现内容简单一致：

```
enter a doc_id to browse:
16745
Happy Birthday Ron Weasley ;;)
enter a doc_id to browse:
16787
HAPPY BIRTHDAY RON WEASLEY !!!
enter a doc_id to browse:
17011
Happy birthday ron weasley
enter a doc_id to browse:
```

这也解释了为何会有许多打分高且相等的 doc 的现象。

5. 检索评价

在完成检索任务后，需要对检索效果进行评价，评价的主要方法是对 queryId 为 171 到 225 的检索内容进行检索，将检索结果与 groundtruth 进行对比，计算评价指标 MAP，NDCG. 因此输入的应是 queryId (171~225)，输出结果不再是 doc_id，而是 tweetId，并且需要按指定格式输出。因此对之前的的检索过程进行以下改进：

1. 引入 tweetId

   之前读取 tweet 时用的是文本读取，而在后续评价时需要用到 tweetId，这里改为用 json 格式读取；

2. 检索并对结果进行排序

   下载 TREC 2014 test topics 文档并将其 queryId 与检索内容对应写入 querynumtoquery.txt,接着读入这个文档，并将其处理为字典 num_query.

   依次对 queryID（171~225），根据字典 num_query 检索，将返回的 tweetId 按其评分从高到低排序，由于评价时最多只取 100 个检索返回结果进行评价，这里最多只返回前 100 个 tweetId.

3. 写 result

   将上一步得到的结果按每行 queryId，tweetId 的格式依次写入 result.txt.

4. 评价

   将得到的 result.txt 及 groudtruth 文档 qrels.txt 输入 eval_hw4.py 得到评价结果如下:

	MAP	NDCG
PLNVSM	0.5270	0.6824
BM25	0.5216	0.6778

实验五 聚类

```
import json
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.cluster import normalized_mutual_info_scor
e
```

读取 tweets 数据集

In [3]:

```
tweets_text = []    #tweets_text 中的元素为原每个 tweet 的正文
clusterId = []      #每个 tweet 实际所在的 cluster
with open('Homework5Tweets.txt', encoding='utf-8') as file:
    for line in file:
        tweet = json.loads(line)
        tweets_text.append(tweet['text'])
        clusterId.append(tweet['cluster'])
# print(tweets_text)
# print(clusterId)
```

建立基于 Tfidf 的向量空间模型

In [4]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
tweets_str = [" ".join(tweets_text)]
# print(tweets_str)
# 创建 transform
vectorizer = TfidfVectorizer()
# 分词并建立词汇表
vectorizer.fit(tweets_text)
# 结果输出
# print(vectorizer.vocabulary_)

vector = vectorizer.transform(tweets_text)
vs_array = vector.toarray()
# 输出编码后的向量信息
print(vector.shape)
print(type(vector))
print(vs_array)
```

```
(2472, 5097)
<class 'scipy.sparse.csr.csr_matrix'>
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

聚类算法测试

In [6]:

```
import numpy as np
X = np.array(vs_array)
print(X)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

K-means

In [7]:

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=110, random_state=0).fit(X)
print(kmeans.labels_)
print(len(kmeans.labels_))
# for i in kmeans.labels_:
#     print(i)
```

```
[105  20  98 ...  52  38  11]
2472
```

k-means 结果评价

In [92]:

```
nmi_kmeans = normalized_mutual_info_score(clusterId, kmeans.lab
els_)
print('nmi_kmeans = ',nmi_kmeans)
```

```
nmi_kmeans =  0.7980301635709245
```

AffinityPropagation

In [93]:

```
from sklearn.cluster import AffinityPropagation
clustering_ap = AffinityPropagation().fit(X)
print(clustering_ap)
print(clustering_ap.labels_)
```

```
AffinityPropagation(affinity='euclidean', convergence_iter=15,
copy=True,
```

```
        damping=0.5, max_iter=200, preference=None, verbose=Fal
se)
[205  91 216 ... 224 145 115]
```

AffinityPropagation 结果评价

In [94]:

```
nmi_ap = normalized_mutual_info_score(clusterId, clustering_ap.
labels_)
print('nmi_ap = ',nmi_ap)
nmi_ap =  0.7831643850554082
```

Mean-shift

In [87]:

```
from sklearn.cluster import MeanShift
clustering_ms = MeanShift(bandwidth=2).fit(X)
print(clustering_ms)
print(clustering_ms.labels_)
MeanShift(bandwidth=2, bin_seeding=False, cluster_all=True, min
_bin_freq=1,
     n_jobs=1, seeds=None)
[0 0 0 ... 0 0 0]
```

Mean-shift 结果评价

In [95]:

```
nmi_ms = normalized_mutual_info_score(clusterId, clustering_ms.
labels_)
print('nmi_ms = ',nmi_ms)
nmi_ms =  -1.6132928326584306e-06
```

Spectral clustering

In [96]:

```
from sklearn.cluster import SpectralClustering
clustering_sc = SpectralClustering(n_clusters=110, assign_label
s="discretize", random_state=0).fit(X)
print(clustering_sc)
print(clustering_sc.labels_)
SpectralClustering(affinity='rbf', assign_labels='discretize',
coef0=1,
          degree=3, eigen_solver=None, eigen_tol=0.0, gamma=1.0,
          kernel_params=None, n_clusters=110, n_init=10, n_jobs=
1,
          n_neighbors=10, random_state=0)
```

```
[21  9  0 ... 81 53 54]
```

spectral clustering 结果评价

In [97]:
```
nmi_sc = normalized_mutual_info_score(clusterId, clustering_sc.
labels_)
print('nmi_sc = ',nmi_sc)
nmi_sc =  0.7800448149457381
```

ward hierarchical clustering

合并于 AgglomerawtiveClustering 中，默认 linkage='ward'

In [98]:
```
from sklearn.cluster import AgglomerativeClustering
clustering_wh = AgglomerativeClustering(affinity='euclidean', c
ompute_full_tree='auto',
            connectivity=None, linkage='ward', memory=None, n_clu
sters=110).fit(X)
print(clustering_wh)
print(clustering_wh.labels_)
AgglomerativeClustering(affinity='euclidean', compute_full_tree
='auto',
            connectivity=None, linkage='ward', memory=None, n_clu
sters=110,
            pooling_func=<function mean at 0x000001FB5269A9D8>)
[ 30  46   2 ...  62  44 102]
```

ward hierarchical clustering 结果评价

In [99]:
```
nmi_wh = normalized_mutual_info_score(clusterId, clustering_wh.
labels_)
print('nmi_wh = ',nmi_wh)
nmi_wh =  0.77587403569932
```

AgglomerativeClustering

In [100]:
```
from sklearn.cluster import AgglomerativeClustering
clustering_agg1 = AgglomerativeClustering(affinity='euclidean',
 compute_full_tree='auto',
            connectivity=None, linkage='complete', memory=None, n
_clusters=110,
```

```
          ).fit(X)
clustering_agg2 = AgglomerativeClustering(affinity='euclidean',
 compute_full_tree='auto',
          connectivity=None, linkage='average', memory=None, n_
clusters=110,
          ).fit(X)
```

AgglomerativeClustering 结果评价

In [101]:

```
nmi_agg_complete = normalized_mutual_info_score(clusterId, clus
tering_agg1.labels_)
nmi_agg_average = normalized_mutual_info_score(clusterId, clust
ering_agg2.labels_)
print('nmi_agg_complete = ',nmi_agg_complete)
print('nmi_agg_average = ',nmi_agg_average)
nmi_agg_complete =  0.7595908816134626
nmi_agg_average =  0.9004539868135747
```

DBSCAN

In [108]:

```
from sklearn.cluster import DBSCAN
clustering_DBSCAN = DBSCAN(eps=0.02, min_samples=5).fit(X)
print(clustering_DBSCAN)
print(clustering_DBSCAN.labels_)
DBSCAN(algorithm='auto', eps=0.02, leaf_size=30, metric='euclid
ean',
    metric_params=None, min_samples=5, n_jobs=1, p=None)
[-1 -1 -1 ... -1 -1 -1]
```

DBSCAN 结果评价

In [109]:

```
nmi_dbscan = normalized_mutual_info_score(clusterId, clustering
_DBSCAN.labels_)
print('nmi_dbscan = ',nmi_dbscan)
nmi_dbscan =  0.0914960470508469
```

Gaussian mixture

In [112]:

```
from sklearn.mixture import GaussianMixture
model = GaussianMixture(n_components = 5, covariance_type = 'ful
l')
model.fit(X)
```

```
label_pred = model.predict(X)

print(label_pred)
```
```
[0 0 0 ... 0 0 0]
```

Gaussian mixture 结果评价

In [113]:

```
nmi_gm = normalized_mutual_info_score(clusterId, label_pred)
print('nmi_gm = ',nmi_gm)
```
```
nmi_gm =  0.4506933941721244
```