基于 sklearn 的多种聚类算法的实现与评测

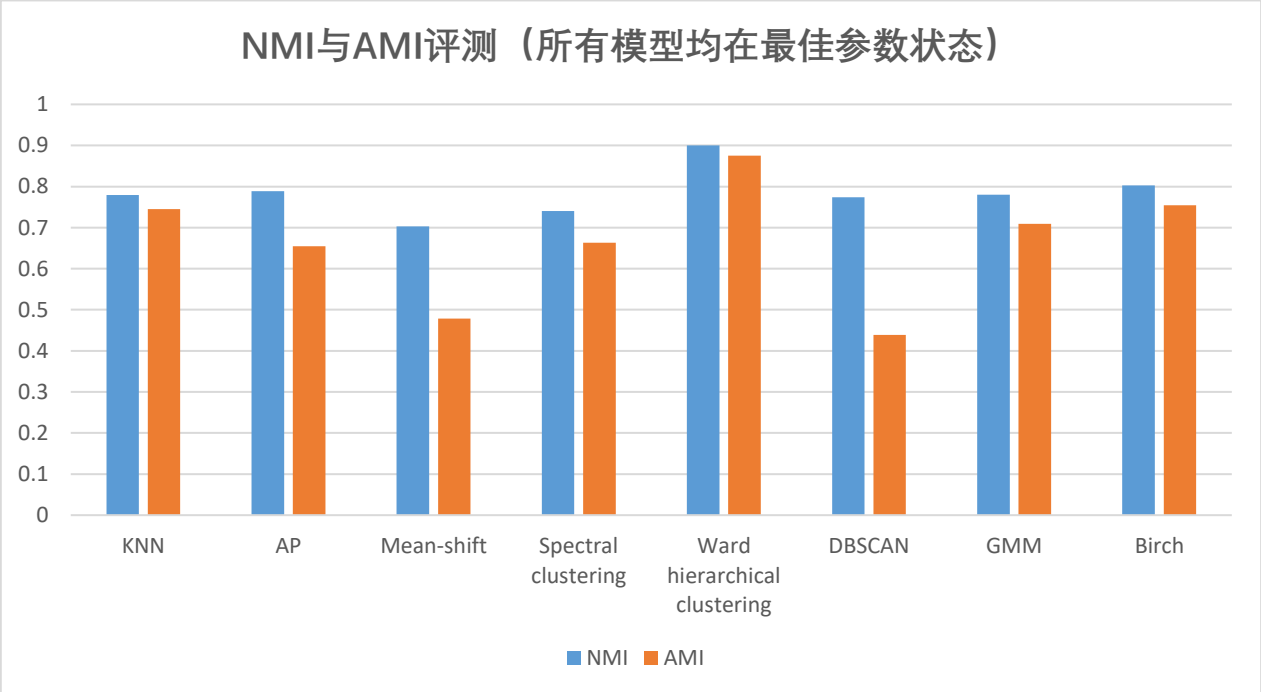
{ **姓名**：孟川；**学号**：201814828；**班级**：2018 级学硕班；**导师**：陈竹敏 }

摘要：通过 sklearn 在 Tweets 数据集上应用多种聚类方法，我主要发现了以下结论：

1. Ward hierarchical clustering、Birch 与 Gaussian mixture models 三个模型非常适合用于文本内容的聚类，并且三个模型时间效率较高；
2. Spectral clustering 模型在时间效率上表现地最好；
3. 密度型聚类方法（例如 DBSCAN 与 Mean-shift）往往时间效率比较低，并且在 NMI 与 AMI 两个评测指标上差距较大（AMI 上表现分值更低）。

NMI 与 AMI 评测（所有模型均在最佳参数状态）

	NMI	**AMI**	**运行时间(s)**
KNN	0.779365	0.745205	58.606697
Affinity Propagation	0.788888	0.654406	36.910559
Mean-shift	0.703239	0.478830	69.337893
Spectral clustering	0.740351	0.663427	**3.623575**
Ward hierarchical clustering	**0.899808**	**0.875384**	22.521459
DBSCAN	0.773610	0.439140	77.088685
Gaussian mixture models	0.780075	0.709347	8.277973
Birch	0.803051	0.754462	17.408954



NMI 与 AMI 评测（所有模型均在最佳参数状态）

目录

一、 Tweets 数据集预处理

预处理部分由两个部分构成：1.操作之前对数据集进行信息统计；2.得到数据集 document 的 tf-idf 表示，并保存。

1.1 数据集统计信息

数据集统计部分主要有两部分构成，分别是基本信息统计与 cluster label 的频率统计。

1.1.1 基本信息统计

以下是数据集的基本信息统计。

表 1 数据集基本信息统计

Document 数量	Cluster label 数量
2472	89

1.1.2 cluster label 的频数统计

我们发现该数据集 cluster label 的频率分布差距比较大，有些 cluster label 频数很大，有的 cluster label 频数很小。详细情况见图 1。

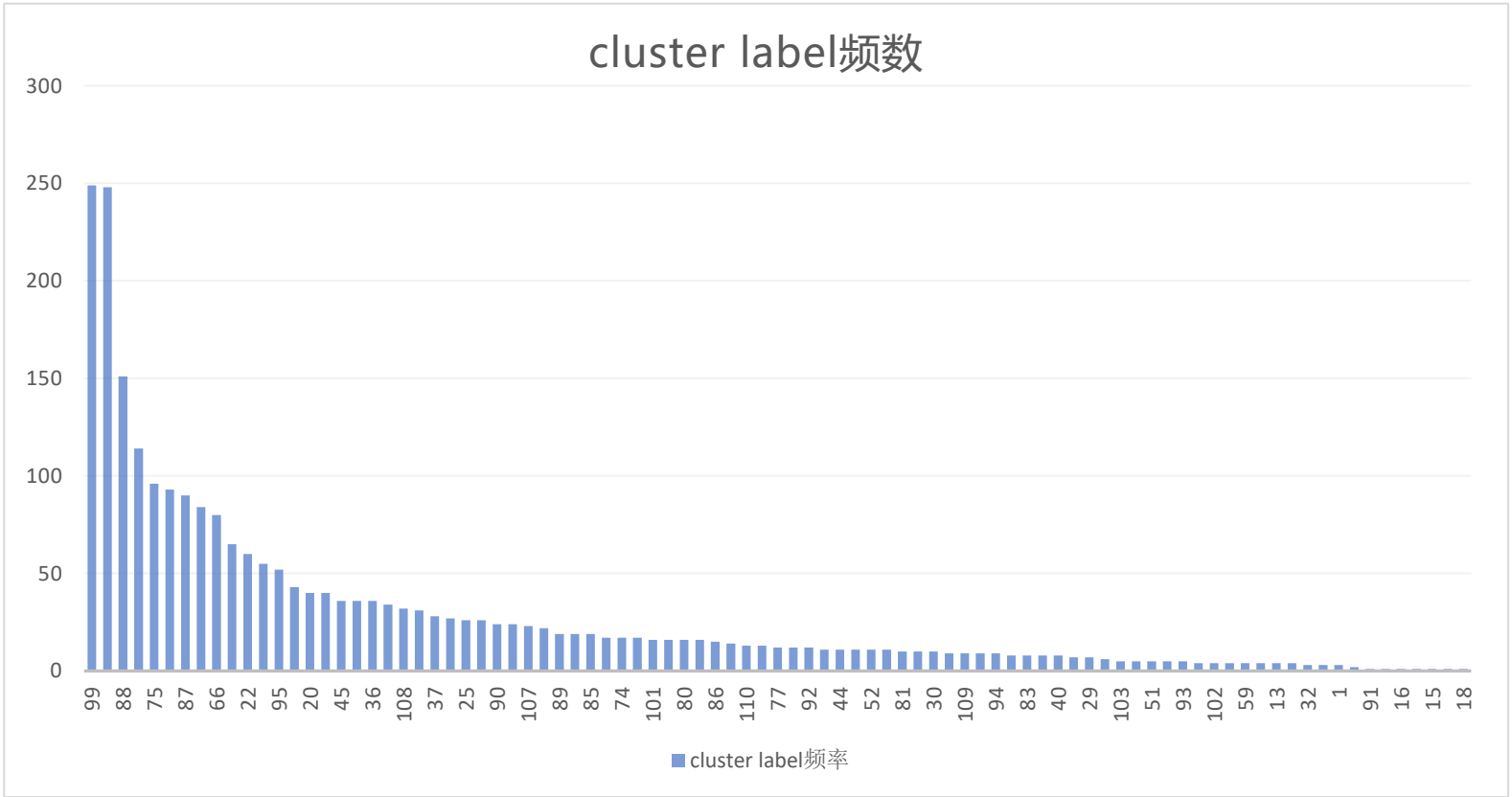


图 1 cluster label 频数统计(cluster label 按频数由大到小的顺序从左至右依次排开)

从图 1 可得，排名比较高的三个 cluster label 99、60、88 别对应的频数是 249、248、151，**而排名后 19 个 cluster label 的频数均小于 5**，排名后 7 个 cluster label 的频数均为 1。所以我们得知，虽然该数据集的 cluster label 总数为 89，但是很可能频数较小的 cluster label 在聚类中充当了噪音角色。

1.2 得到 tf-idf 嵌入

先得到文档的 tf-idf 嵌入与相应的 cluster label，然后将其保存。

1.2.1 调用 sklearn 封装的 tf-idf 方法

调用 sklearn 的 CountVectorizer、vectorizer.fit_transform、transformer.fit_transform 等方法，构建长度为 5097 的字典，最终得到所有 document 的 tf-idf 表示矩阵，矩阵的维度为2472×5097。

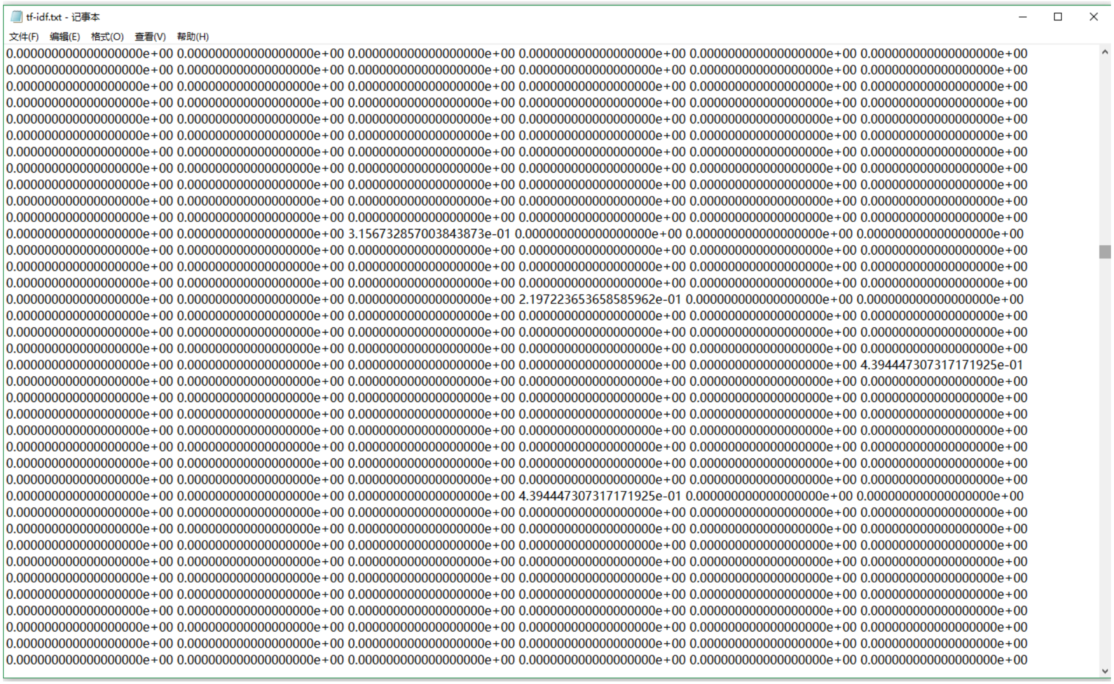
```
16
17      #下面使用sklearn调用tf-idf获得每个utterance的embeding
18      vectorizer = CountVectorizer()
19      count = vectorizer.fit_transform(corpus)
20      transformer = TfidfTransformer()
21      tfidf_matrix = transformer.fit_transform(count)
22      np.savetxt("embeding/tf-idf.txt", tfidf_matrix.toarray())
23
```

图 2 使用 sklearn 的 tf-idf 方法

图 3 字典中的词

1.2.2 embeding 与 label 的保存

将 document 的 embeding 矩阵和相应的 label 保存，方便后续使用。



tf-idf.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 3.156732857003843873e-01 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 2.197223653658585962e-01 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 4.394447307317171925e-01
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 4.394447307317171925e-01 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00
```

图 4 tf-idf 嵌入矩阵

二、 调用 sklearn 聚类方法与调参

因为 AMI 比 NMI 在偶然情况处理更合理，相对更合理一些，本文模型调参时均用 AMI 作为评测指标。

2.1 KNN

KNN 的核心代码如下。

```
def k_means(data):
    t0 = time()
    estimator = KMeans(n_clusters=50)#构造聚类器
    estimator.fit(data)#聚类
    label_pred = estimator.labels_ #获取聚类标签
    time_use=time()-t0
    return label_pred,time_use
```

图 5 KNN 核心代码

注意 Tweets 数据集的 label 分类个数是 89，所以我们也实现调整 KNN 的聚类个数为 89（n_clusters=89），但是发现，当 **n_clusters=70** 的时候，实验结果最好,AMI 取得最大值 0.745205，这可能就是**因为 cluster label 排序 70 位之后的 document 数量太少，反而成了噪音干扰了其他正常的聚类**。

表 2 KNN 调参表

n_clusters	AMI	运行时间(s)
89	0.679438	73.929518
80	0.709297	65.360532
70	**0.745205**	**58.606697**
60	0.727317	54.959479
50	0.650711	46.455092

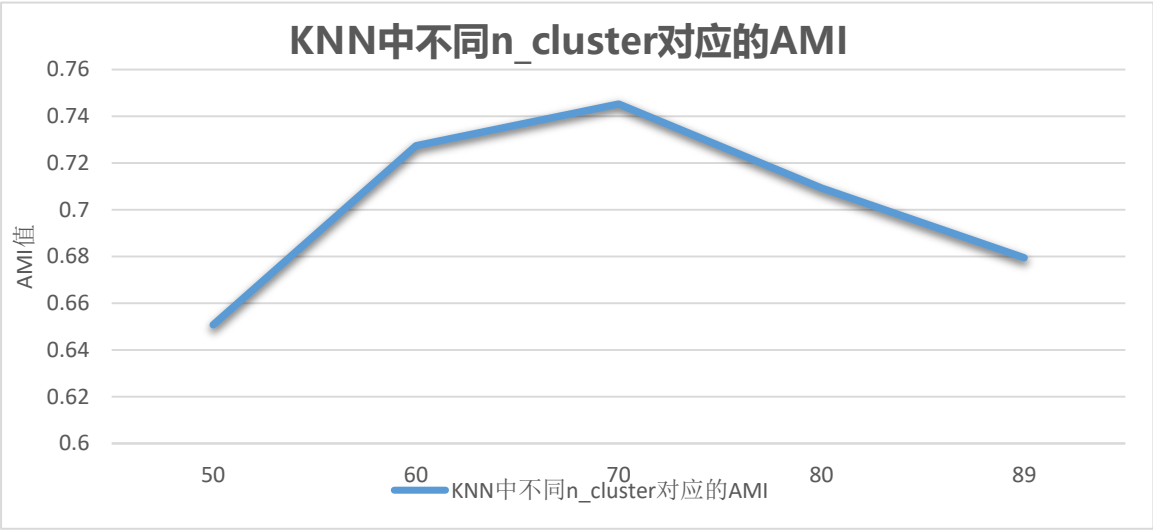


图 6 KNN 中不同 n_cluster 对应的 AMI

2.2 Affinity Propagation(AP)

Affinity Propagation(AP)的核心代码如下。

```
def Affinity_Propagation(data):
    af = AffinityPropagation(preference=-5.74).fit(data)
    cluster_centers_indices = af.cluster_centers_indices_
    label_pred= af.labels_
    n_clusters= len(cluster_centers_indices)
    print(n_clusters)
    return label_pred
```

图 7 Affinity Propagation 核心代码

实验发现在默认情况下，AP 聚类方法非常倾向于得到多个聚类结果（当“preference”=None 时，得到了 320 个类）。而 Tweets 数据集一共才 89 个类，必须调整“preference”来降低每个 point 是聚类中心点的可能性，从而降低聚类所得的类别个数，最终发现当**“preference”=-5.74 时，model 的 AMI 指标最高，AMI 达到 0.654407，运行时间为 36.910559**s。详情见表 3。

表 3 AP 模型的调参表（1）

Preference 值	**聚类类别个数**	**AMI**
0	2398	0.010602344528155293
-1	1603	0.1616498679626918
-2	320	0.4896257947296034
-3	218	0.5630028005415774
-4	142	0.6180687349737378
-5	115	0.6287995540653826
-5.5	104	0.6395386600529404
-5.6	117	0.6388484325316311
-5.7	98	0.64935267412474
-5.74	**98**	**0.6544066133519121**
-5.8	2472	0.000000

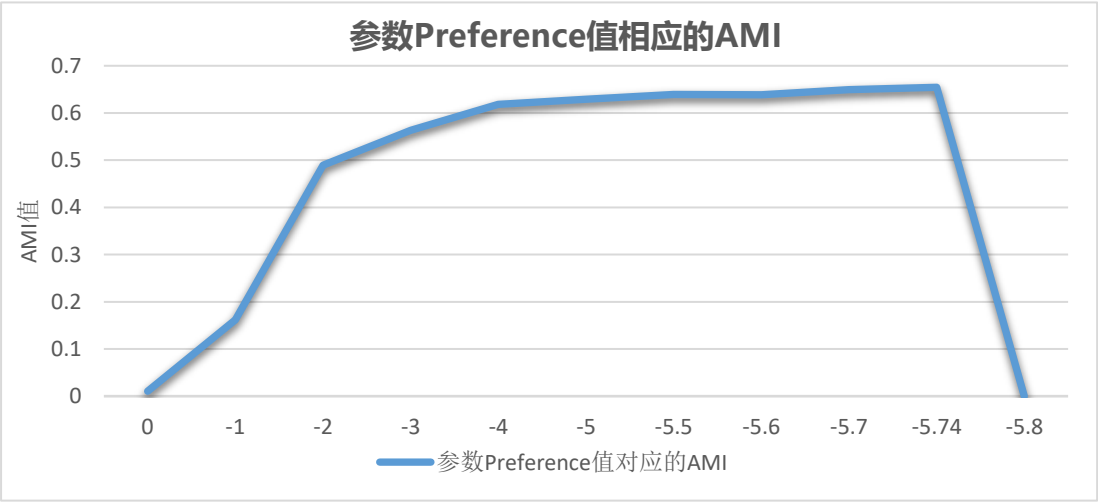


图 8 AP 模型的调参

我们在“preference”=-5.74 的情况下，**改变 damping（阻尼系数）的值，观察实验结果发现并没有得到提升。**

表 4 AP 模型的调参表（2）

damping 值	聚类类别个数	AMI
0.6	97	0.652058
0.7	97	0.648307
0.8	96	0.647118
0.9	97	0.649740

2.3 Mean-shift

Mean-shift 的预测核心代码如下。

```
def Mean_shift(data):
    t0 = time()
    bandwidth=0.9
    ms = MeanShift(bandwidth=bandwidth,bin_seeding=True,cluster_all=True)
    ms.fit(data)
    label_pred = ms.labels_
    labels_unique = np.unique(label_pred)
    n_clusters = len(labels_unique)
    time_use=time()-t0

    print("bandwidth参数设置为: ",bandwidth)
    print("得到的cluster数目为: ",n_clusters)
    return label_pred,time_use
```

图 9 Mean-shift 核心代码

当 bin_seeding=False 的时候，调整 Bandwidth，得到 AMI 的值见表 5。发现参数“bandwidth”设置为 0.9 的时候，AMI 取得最大值。

表 5 Mean-shift 调参表（1）

bandwidth 值	聚类类别个数	AMI
1	1148	0.12148586730363102
0.9	1634	0.20377716278710623
0.8	1976	0.10020146997456376
0.7	2119	0.0635485205329769
0.6	2208	0.04422927942354482
0.5	2272	0.032440

当 bin_seeding=True 的时候，调整 Bandwidth，得到 AMI 的值见表 6。

表 6 Mean-shift 调参表（2）

bandwidth 值	聚类类别个数	AMI	运行时间(s)
1	299	0.014363	138.083952
0.9	478	0.478830	69.337893
0.8	605	0.377089	36.676130
0.7	696	0.347504	34.564874
0.6	871	0.312399	27.771711
0.5	1257	0.209903	32.459113

整体而言，当 bin_seeding 设置为 true 时，比设置为 False 时提升幅度较大，并且仍旧在“bandwidth”设置为 0.9 的时候 AMI 取得最大值，最大值为 **0.478830，运行时间为 69.337893s。**

2.4 Spectral clustering

以下为 Spectral clustering 的核心代码。

```
54 def Spectral_clustering(data):
55     sc=SpectralClustering(gamma=0.3,n_clusters=89).fit(data)
56     label_pred =sc.labels_
57 
58     return label_pred
```

图 10 Spectral clustering 核心代码

当 **n_clusters=89** 的时候，gamma 值的改变引起 AMI 的改变，详见表 7。我们发现当 gamma 值取 0.5 的时候，AMI 取得最大值 **0.658281，运行时间为 4.583276s。**gamma 值取大于 1 的时候，结果普遍较差，在此不列出结果。

表 7 Spectral clustering 调参表（1）

gamma 值	**AMI**	**运行时间(s)**
1（默认）	0.555137	4.558172
0.9	**0.552800**	**4.549476**
0.8	0.610997	4.330443
0.7	0.638527	4.430617
0.6	0.630709	4.198558
0.5	**0.658281**	**4.583276**
0.4	0.633030	4.442534
0.3	0.649792	4.060393
0.2	0.643518	4.314387
0.1	0.646111	4.203846

当 **n_clusters=70** 的时候，gamma 值的改变引起 AMI 的改变，详见表 8。我们发现当 gamma 值为 0.1 的时候，AMI 取得最大值 **0.663427，时间为 3.623575s**，大于在 n_clusters=89 的最优值。

表 8 Spectral clustering 调参表（2）

gamma **值**	**AMI**	**运行时间(s)**
0.8	0.584451	4.088376
0.7	0.595617	3.740295
0.6	0.600795	3.495676
0.5	0.606832	3.792112
0.4	0.612978	4.347775
0.3	0.641143	3.672531
0.2	0.657407	3.668014
0.1	**0.663427**	**3.623575**

2.5 Ward hierarchical clustering

以下为 Ward hierarchical clustering 的核心代码。

```
def Ward_hierarchical_clustering(data):
    t0 = time()
    whc = AgglomerativeClustering(affinity='euclidean',n_clusters=80, linkage='average').fit(data)
    label_pred=whc.labels_
    time_use=time()-t0

    return label_pred,time_use

```

图 11 Ward hierarchical clustering 的核心代码

当 **n_clusters=89** 的时候，linkage 值的改变引起 AMI 的改变，详见表 9。我们发现当 linkage 值取 **average** 的时候，AMI 取得最大值 **0.8682575924833005，运行时间为 24.077788s。** gamma 值取大于 1 的时候，结果普遍较差，在此不列出结果。

表 9 Ward hierarchical clustering 调参表（1）

linkage	**AMI**	**运行时间(s)**
Ward	0.6948668794233028	22.870451
complete	0.633430556622537	22.734272
average	**0.8682575924833005**	**24.077788**

我们固定 linkage 取值 average，测试该方法在选取不同 clusters 时的表现，详见表 10，我们发现当 n_clusters 取 80 的时候，AMI 取得最大值 **0.875384**，运行时间 22.521459，最优值优于 n_clusters=89 的情况。

表 10 Ward hierarchical clustering 调参表（2）

n_clusters	**AMI**	**运行时间(s)**
85	0.873396	23.018158
80	**0.875384**	**22.521459**
70	0.834434	22.583766
60	0.817438	22.419841

当 affinity 取除 euclidean 之外的 manhattan 或 cosine 时，最优结果均不如取 euclidean 的情况，在此实验结果略去。

2.5 DBSCAN

DBSCAN 的核心代码如下。

```
81
82 def DBSCAN_(data):
83     t0 = time()
84     db=DBSCAN(eps = 1.1, min_samples = 1).fit(data)
85     label_pred=db.labels_
86     time_use=time()-t0
87     n_clusters = len(set(label_pred))
88     print("聚类得到的cluster数目为：",n_clusters)
89     return label_pred,time_use
90
```

图 12 DBSCAN 的核心代码

我们在固定参数 eps 为 1 的情况下，调整参数 min_samples 使得 AMI 最大化，详情见表 11。最终我们发现当 eps 为 1，min_samples 为 2 的情况下 AMI 取得最大值 **0.368993**，运行时间 76.109303s。

表 11 DBSCAN 调参表（1）

eps	min_samples	AMI	运行时间(s)
1	1	0.272834	75.518468
1	**2**	**0.368993**	**76.109303**
1	3	0.316936	75.943875
1	4	0.266976	76.912437.

我们在固定参数 min_samples 为 1 的情况下，调整参数 eps 使得 AMI 最大化，详情见表 12。最终我们发现当 eps 为 1.1, min_samples 为 1 的情况下, AMI 取得最大值 **0.439140**，运行时间为 77.088685，此结果要优于参数表 9 所探讨的参数设置。

表 12 DBSCAN 调参表（2）

eps	min_samples	AMI	运行时间(s)
0.5	1	0.032440	9.223348
0.6	1	0.042392	11.669970
0.7	1	0.061659	16.524368
0.8	1	0.088532	25.970736
0.9	1	0.151589	42.258640
1.1	**1**	**0.439140**	**77.088685**
1.2	1	0.209495	75.698330
1.3	1	0.000528	76.858662
1.4	1	0.000000	78.167717
1.5	1	0.000000	75.911795

2.5 Gaussian mixtures

Gaussian mixture models 的核心代码如下。

```
92 def Gaussian_mixtures(data):
93     t0 = time()
94     gmm=mixture.GaussianMixture(n_components=70,covariance_type='spherical').fit(data)
95     label_pred=gmm.predict(data)
96     time_use=time()-t0
97
98     return label_pred,time_use
```

图 13 Gaussian mixture models 核心代码

由于 Gaussian mixture models 的参数 covariance_type 设置为 full、tied 与 diag 时，运行速度非常慢，甚至发生内存溢出的情况，所以本次实验主要考虑 covariance_type 为 spherical 的情况，详情见表 13，AMI 随着 n_components 的变化而变化，最终当 n_components 取值为 70 的时候，AMI 取得最大值 **0.709347，运行时间 8.277973s**。

表 13 Gaussian mixtures 调参表（1）

n_components	**AMI**	**运行时间(s)**
6	0.267917	2.418555
7,	0.188810	2.565127
8	0.285478	2.127326
10	0.311632	2.740238
20	0.473211	4.238744
30	0.537531	6.234005
40	0.588370	6.077356
50	0.623506	6.145651
60	:0.695922	7.609859
70	**0.709347**	**8.277973**
80	0.671385	9.624450
89	0.680410	8.699822

我们再设置 covariance_type 为 tied，测试模型在 n_components 取值为 70 与 89 时的表现，见表 14 。我们发现当 n_components 取值为 70 的时候，AMI 取得最大值 **0.717415，运行时间为 175.478829。**

表 14 Gaussian mixtures 调参表（2）

n_components	**AMI**	**运行时间(s)**
70	**0.717415**	**175.478829**
89	0.667417	225.105440

2.5 Birch

Birch 的核心代码如下。

```
def birch(data):
    t0 = time()
    label_pred = Birch(n_clusters = 70,threshold = 0.4, branching_factor =70).fit_predict(data)
    time_use=time()-t0

    return label_pred,time_use

```

图 14 Birch 核心代码

当 n_clusters = 89 时，我们调整模型参数 threshold、branching_factor 似 NM1 最大化，详见表 15 我们发现当 threshold 取 0.4，branching_factor 取 50 的时候，AMI 达到最大值 **0.709309**，运行时间为 18.888673。

表 15 Birch 调参表（1）

threshold	branching_factor	AMI	运行时间(s)
0.4	40	0.708061	18.727634
0.1	50	0.689846	22.979831
0.2	50	0.690894	22.652373
0.3	50	0.698443	19.757403.
0.4	**50**	**0.709309**	**18.888673**
0.5	50	0.703362	15.791411
0.6	50	0.707769	12.468891
0.7	50	0.701796	9.786468
0.8	50	0.687208	7.231609
0.9	50	0.449066	5.142075

当 n_clusters = 70 时，我们调整模型参数 threshold、branching_factor 似 NM1 最大化，详见表 16。我们发现当 threshold 取 0.4，branching_factor 取 70 的时候，AMI 达到最大值 **0.754462**，运行时间为 17.408954，大于在 n_clusters = 89 时的最优值。

表 16 Birch 调参表（2）

threshold	branching_factor	AMI	运行时间(s)
0.1	50	0.722500	21.469722
0.5	50	0.727832	14.994222
0.4	40	0.743386	18.974495
0.4	60	0.745756	17.758003
0.4	**70**	**0.754462**	**17.408954**
0.4	80	0.743896	17.417895
0.4	90	0.737519	17.849248
0.4	100	0.737247	18.029610

三、 聚类结果评测

因为 AMI 比 NMI 在偶然情况处理更合理，相对更合理一些，本文模型调参时均用 AMI 作为评测指标，接下来将会评测所有模型在最佳参数状态下的 NMI 与 AMI 指标评测与运行时间的评测。

3.1 NMI 与 AMI

所有模型均在最佳参数状态的 NMI 与 AMI 结果见表 17 与图 14。

表 17 NMI 与 AMI 评测（所有模型均在最佳参数状态）

	NMI	AMI	运行时间(s)
KNN	0.779365	0.745205	58.606697
Affinity Propagation	0.788888	0.654406	36.910559
Mean-shift	0.703239	0.478830	69.337893
Spectral clustering	0.740351	0.663427	**3.623575**
Ward hierarchical clustering	**0.899808**	**0.875384**	22.521459
DBSCAN	0.773610	0.439140	77.088685
Gaussian mixture models	0.780075	0.709347	8.277973
Birch	0.803051	0.754462	17.408954

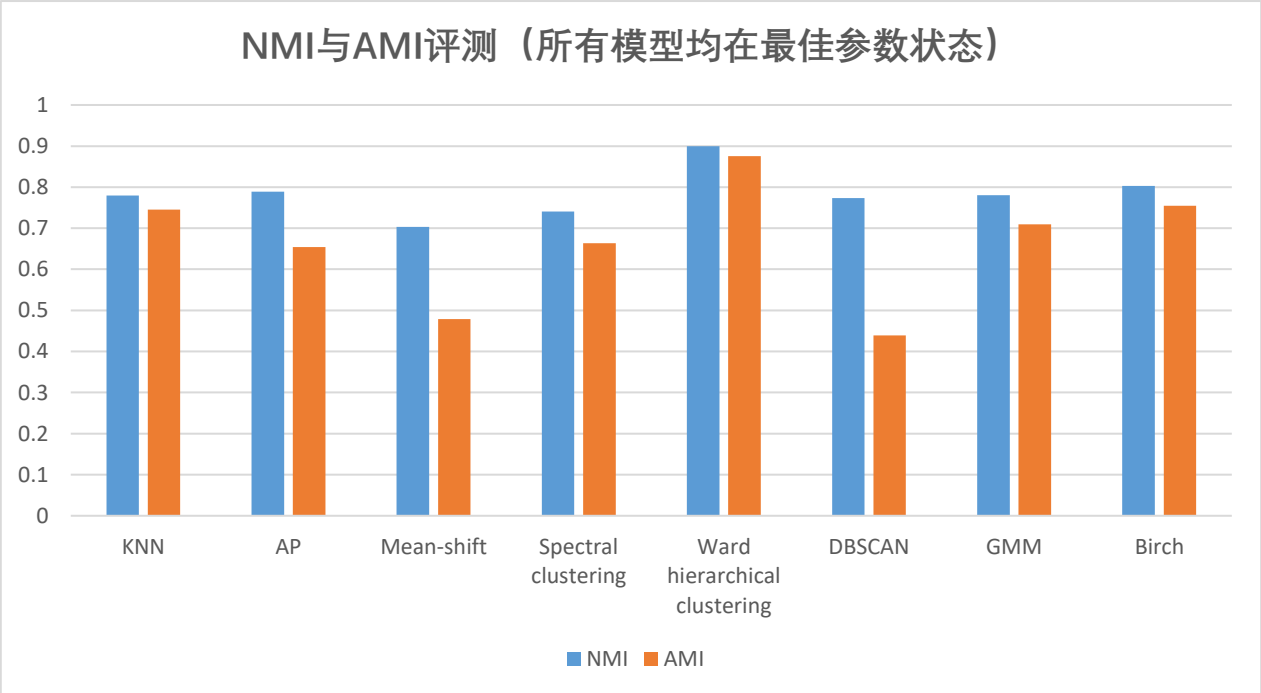


图 15 NMI 与 AMI 评测（所有模型均在最佳参数状态）

由表 17 与图 15 我们发现，**NMI 与 AMI 两个指标基本趋同**（在密度型聚类方法上略有差别），**Ward hierarchical clustering 得到了最高的聚类指标，其次是 Birch 与 Gaussian mixture models，这三个模型表现出对文本内容进行建模的强大能力。**

3.2 运行时间评测

模型在最佳参数状态下的时间评测见图 15。

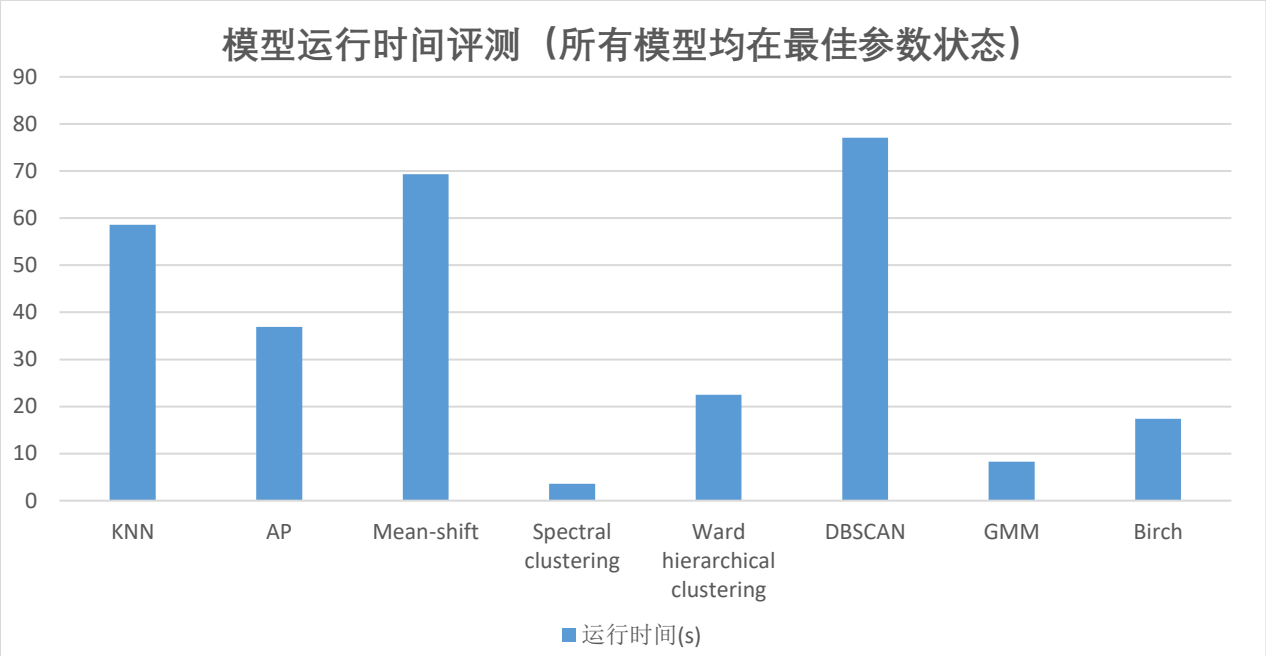


图 16 模型运行时间评测（所有模型均在最佳参数状态）

由图 16 我们可以发现，**Ward hierarchical clustering、 Birch 与 Gaussian mixture models** 三个模型在 **NMI** 与 **AMI** 表现最好的三个模型在**运行时间**上也表现的非常出色，仅次于**运行时间最短的 Spectral clustering。**而 DBSCAN、Mean-shilft、KNN 三个模型运行时间最长，比较耗费时间，这三个模型在 NMI 与 AMI 上的表现也比较一般。

四、 结论

本次聚类算法的实现与评测增强了我的代码能力与对聚类模型的理解。

1. **Ward hierarchical clustering、 Birch 与 Gaussian mixture models** 三个模型非常适合用于文本内容的聚类，并且三个模型时间效率较高。
2. **Spectral clustering 模型在时间效率上表现地最好。**
3. 密度型聚类方法（例如 DBSCAN 与 Mean-shift）往往时间效率比较低，并且在 NMI 与 AMI 两个评测指标上差距较大（AMI 上表现分值更低）。