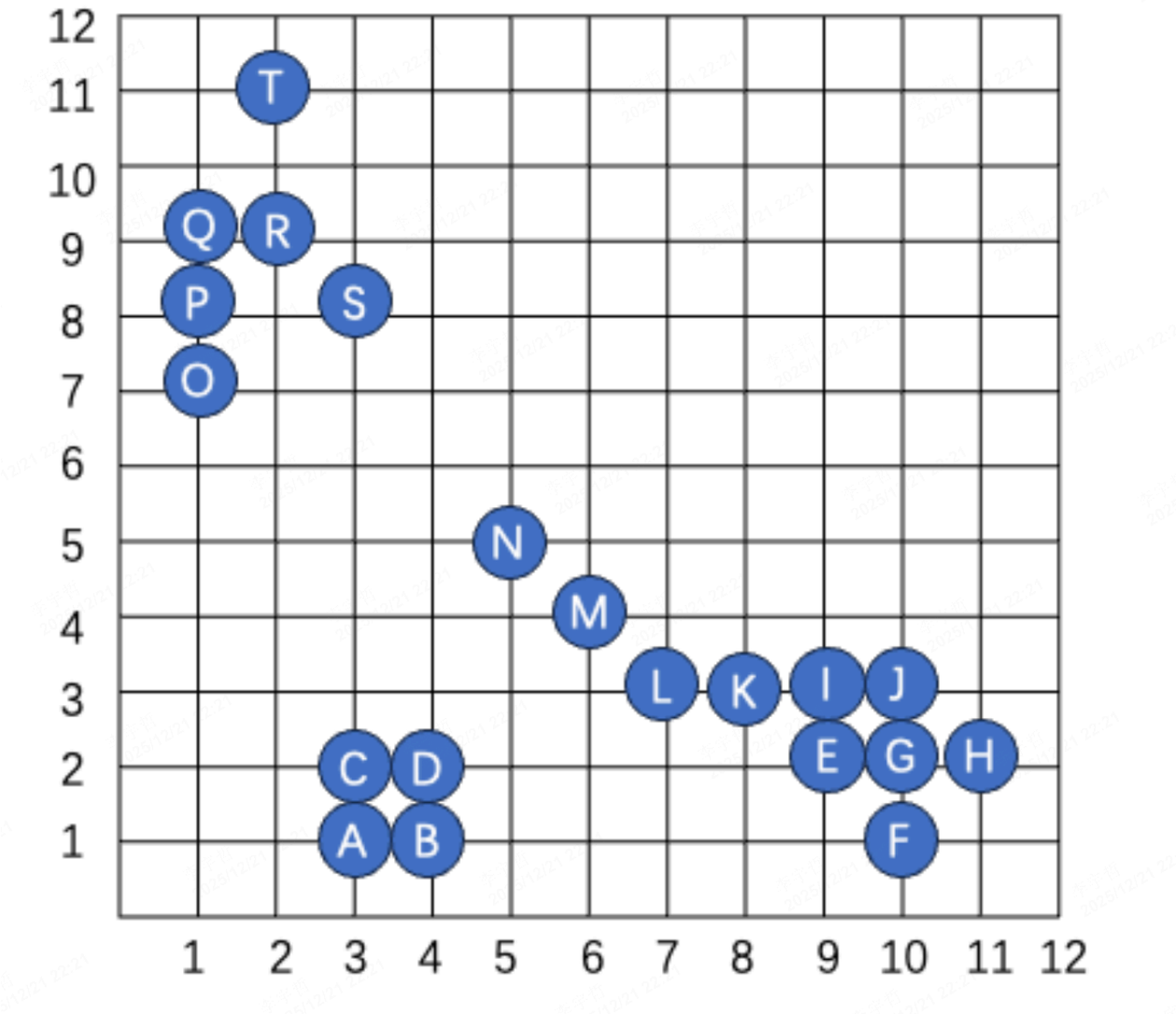


# 计算机应用数学第二次作业

李宇哲 SA25011049

## T1

1、基于如下数据集（二维），分别使用 Manhattan 和 Euclidean 距离，计算 DBSCAN 聚类结果，并标出 core points、border points 和 noise points。使用如下参数 $\epsilon = 1.1$ ,  $minPts = 2$ 。并讨论两种距离的优劣。



Manhattan 距离:  $d_1(p_i, p_j) = |x_i - x_j| + |y_i - y_j|$

Euclidean 距离:  $d_2(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

聚类1，左下角(A,B,C,D)

欧式距离下: A-B=1, A-C=1均临近

A-D= $\sqrt{2} \approx 1.414 > 1.1$  , 不相连

$B-C=\sqrt{2} \approx 1.414 > 1.1$ ，不相连

$C-D=1$ ，邻近

在  $\epsilon = 1.1$  下，每个点至少与2个点以内相连，因此四个点联通，形成一个 cluster

- Core: A,B,C,D
- Cluster 1 = {A,B,C,D}

曼哈顿距离：

$AB=1$ ， $AC=1$ ， $AD=2>1.1$ ， $CD=1$

四点联通

- Core: A,B,C,D
- Cluster 1 = {A,B,C,D}

无 noise point

聚类2, N,M,L,K

欧氏距离下：

$NM=\sqrt{2} \approx 1.414$ ， $ML=\sqrt{2} \approx 1.414$ ， $LK=1$

由于  $\epsilon = 1.1 < 1.414$ ，N与M不会互连，不形成聚类

欧氏距离下无法形成联通簇，曼哈顿距离下也无法形成cluster

因此 N，M，L，K均为 noise points

聚类3, E, F, G, H, I, J

欧氏距离下：

$EF=1.414>1.1$ ， $EG=1$ ， $EI=1$ ， $GH=1$ ， $IJ=1$ ，形成cluster

- Core points = G, E, I, J
- Border: F, H

曼哈顿距离下：

$EG=1$ ， $EI=1$ ， $IJ=1$ ， $GH=1$

- core points = G, E, I, J
- Border = F, H

聚类4: O, P, Q, R, S, T

欧氏距离下：

$OP=1.414>1.1$ ， $PQ=1$ ， $QR=1$ ， $RS=1$ ， $PS=1.414>1.1$ ， $ST=3>1.1$

PQRS形成一个cluster，OT为noist points，core为 QRS

曼哈顿距离下形成相同cluster

cluster	points	core points	Border
C1	A,B,C,D	A,B,C,D	
C2	E,F,G,H,I,J	E,G,I,J	F,H
C3	P,Q,R,S	Q,R,S	P
Noise	O,N,M,L,K,T		

## T2

### 1、高斯混合模型与 EM 算法

数据集：Iris 数据集

数据描述：<https://www.kaggle.com/datasets/uciml/iris>，可通过 sklearn 直接导入数据集

```
from sklearn import datasets
iris = datasets.load_iris()
```

任务描述：使用高斯混合模型与 EM 算法对数据进行分类计算，mixture components 设置为 3。

要求输出：不同高斯分布的 mean 和 variance，每个高斯分布对应的权重，plot 出分布的图。

EM 算法可以参考

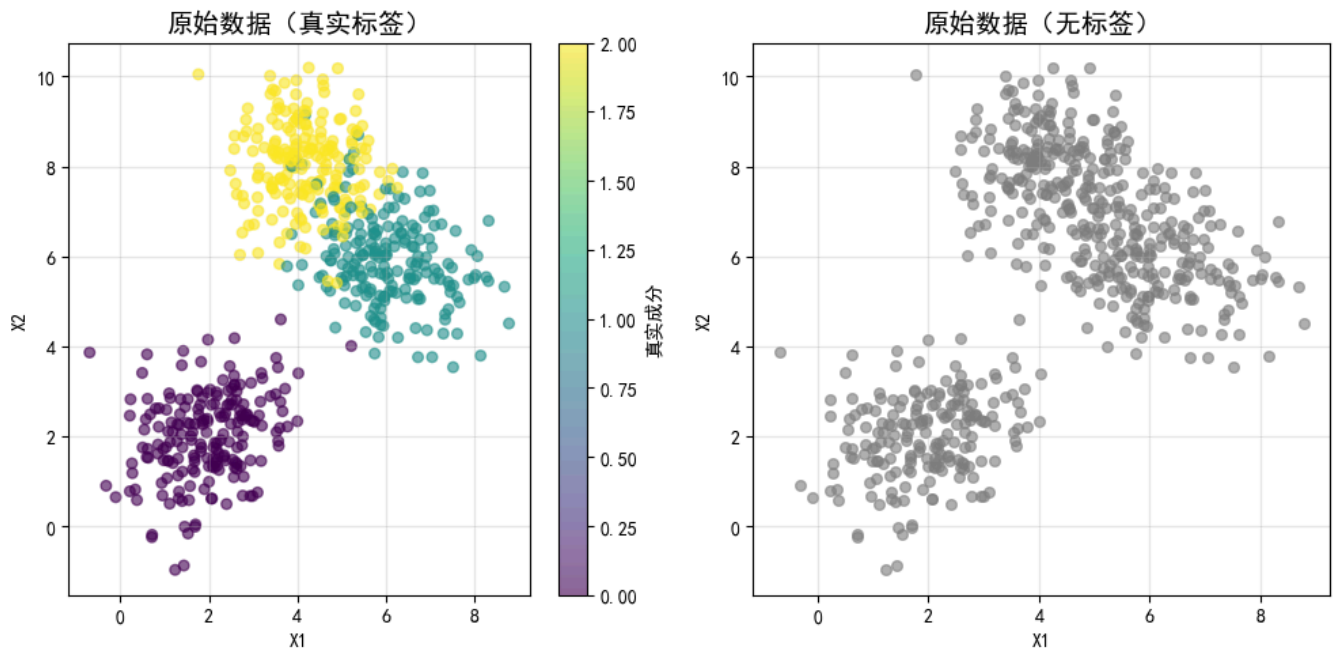
<https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html>。

Optional：尝试不同的 covariance structures，包括 spherical、diagonal、tied 与 full。

定义三个高斯分布

```
1 # 定义三个高斯分布的参数
2 n_samples_per_component = 200
3 n_components = 3
4
5 # 第一个高斯分布
6 mean1 = [2, 2]
7 cov1 = [[1, 0.3], [0.3, 1]]
8 x1 = np.random.multivariate_normal(mean1, cov1, n_samples_per_component)
9
10 # 第二个高斯分布
11 mean2 = [6, 6]
12 cov2 = [[1, -0.3], [-0.3, 1]]
13 x2 = np.random.multivariate_normal(mean2, cov2, n_samples_per_component)
14
15 # 第三个高斯分布
16 mean3 = [4, 8]
17 cov3 = [[0.8, 0], [0, 0.8]]
18 x3 = np.random.multivariate_normal(mean3, cov3, n_samples_per_component)
```

合并之后，数据形状为：(600,2)，一共有三个分布，每个分布的样本数为200



创建并训练 GMM 模型，得到参数如下：

```

1  class GaussianMixtureModel:
2      """
3      高斯混合模型（GMM）使用EM算法进行训练
4
5      参数：
6          n_components: 混合成分的数量（高斯分布的数量）
7          max_iter: 最大迭代次数
8          tol: 收敛阈值
9          random_state: 随机种子
10     """
11
12     def __init__(self, n_components=3, max_iter=100, tol=1e-6, random_state=42):
13         self.n_components = n_components
14         self.max_iter = max_iter
15         self.tol = tol
16         self.random_state = random_state
17         self.weights_ = None
18         self.means_ = None
19         self.covariances_ = None
20         self.log_likelihood_history_ = []
21
22     def _initialize_parameters(self, X):
23         """初始化模型参数"""
24         np.random.seed(self.random_state)
25         n_samples, n_features = X.shape
26         self.weights_ = np.ones(self.n_components) / self.n_components
27         self.means_ = X[np.random.choice(n_samples, self.n_components, replace=False)]
28         cov = np.cov(X.T)
29         self.covariances_ = np.array([cov for _ in range(self.n_components)])
30
31     def _e_step(self, X):
32         """

```

```

33     E步：计算每个样本属于每个高斯分布的后验概率（责任）
34
35     返回：
36         responsibilities: (n_samples, n_components) 形状的数组
37     """
38     n_samples = x.shape[0]
39     responsibilities = np.zeros((n_samples, self.n_components))
40
41     for k in range(self.n_components):
42         try:
43             responsibilities[:, k] = self.weights_[k] * multivariate_normal.pdf(
44                 x, self.means_[k], self.covariances_[k]
45             )
46         except:
47             responsibilities[:, k] = self.weights_[k] * multivariate_normal.pdf(
48                 x, self.means_[k], self.covariances_[k] +
49                 np.eye(self.covariances_[k].shape[0]) * 1e-6
50             )
51
52     responsibilities_sum = responsibilities.sum(axis=1, keepdims=True)
53     responsibilities_sum[responsibilities_sum == 0] = 1e-10
54     responsibilities = responsibilities / responsibilities_sum
55
56     return responsibilities
57
58 def _m_step(self, x, responsibilities):
59     """
60     M步：根据当前的责任更新模型参数
61     """
62     n_samples, n_features = x.shape
63     Nk = responsibilities.sum(axis=0)
64     self.weights_ = Nk / n_samples
65     self.means_ = np.zeros((self.n_components, n_features))
66     for k in range(self.n_components):
67         self.means_[k] = (responsibilities[:, k][:, np.newaxis] * x).sum(axis=0) /
68         Nk[k]
69
70     self.covariances_ = np.zeros((self.n_components, n_features, n_features))
71     for k in range(self.n_components):
72         diff = x - self.means_[k]
73         self.covariances_[k] = np.dot(
74             responsibilities[:, k] * diff.T, diff
75         ) / Nk[k]
76         self.covariances_[k] += np.eye(n_features) * 1e-6
77
78 def _compute_log_likelihood(self, x):
79     """计算对数似然"""
80     n_samples = x.shape[0]
81     log_likelihood = 0
82
83     for k in range(self.n_components):
84         try:
85             log_likelihood += self.weights_[k] * multivariate_normal.pdf(
86                 x, self.means_[k], self.covariances_[k]

```

```

84         )
85     except:
86         log_likelihood += self.weights_[k] * multivariate_normal.pdf(
87             x, self.means_[k], self.covariances_[k] +
88             np.eye(self.covariances_[k].shape[0]) * 1e-6
89         )
90
91     log_likelihood = np.log(log_likelihood + 1e-10).sum()
92     return log_likelihood
93
94 def fit(self, X):
95     """
96     使用EM算法训练模型
97
98     参数:
99     X: 训练数据, 形状为 (n_samples, n_features)
100    """
101    X = np.array(X)
102    self._initialize_parameters(X)
103
104    prev_log_likelihood = -np.inf
105
106    for iteration in range(self.max_iter):
107        responsibilities = self._e_step(X)
108        self._m_step(X, responsibilities)
109        log_likelihood = self._compute_log_likelihood(X)
110        self.log_likelihood_history.append(log_likelihood)
111        if abs(log_likelihood - prev_log_likelihood) < self.tol:
112            print(f"在第 {iteration + 1} 次迭代后收敛")
113            break
114
115        prev_log_likelihood = log_likelihood
116
117    return self
118
119 def predict(self, X):
120     """
121     预测每个样本最可能属于哪个高斯分布
122
123     参数:
124     X: 测试数据
125
126     返回:
127     预测的类别标签
128     """
129    responsibilities = self._e_step(X)
130    return np.argmax(responsibilities, axis=1)
131
132 def predict_proba(self, X):
133     """
134     返回每个样本属于每个高斯分布的概率
135
136     参数:

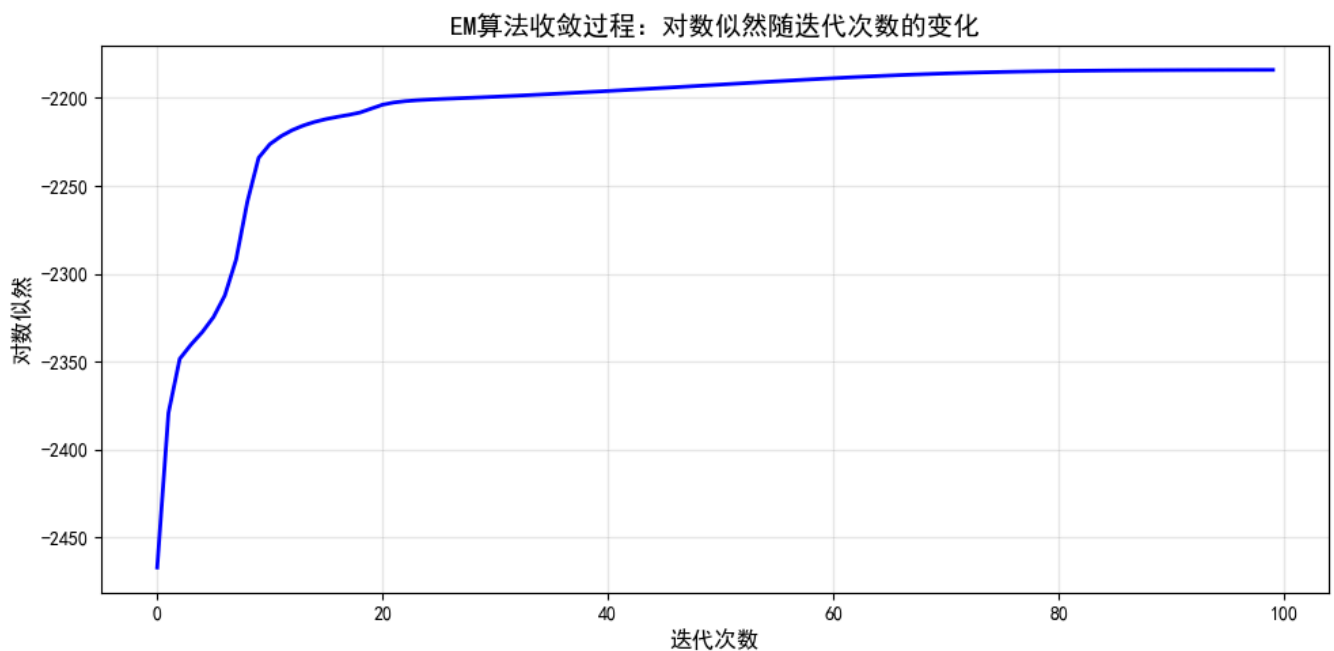
```

```
136         x: 测试数据
137
138     返回:
139         概率矩阵, 形状为 (n_samples, n_components)
140         .....
```

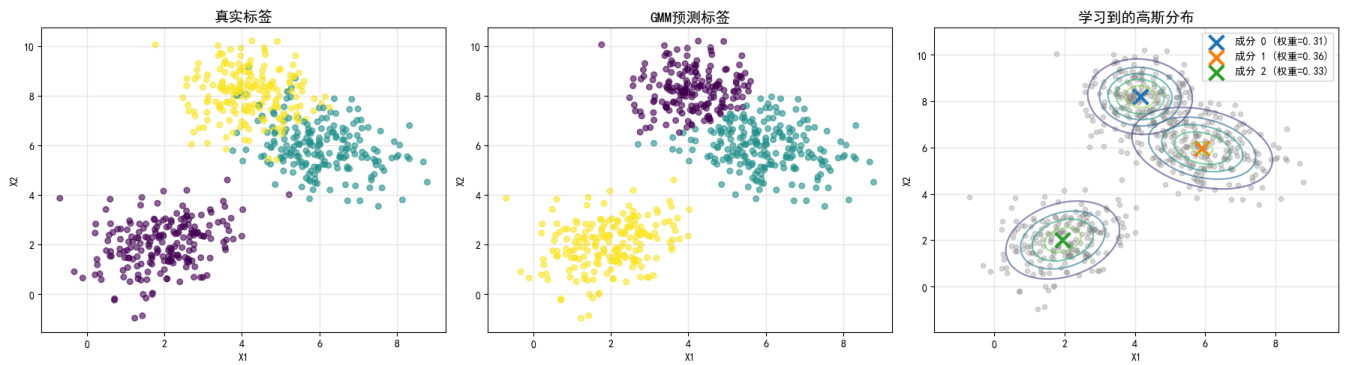
```
141     return self._e_step(x)
142
```

```
1  混合权重: [0.30550761 0.36355525 0.33093714]
2
3  均值:
4      成分 0: [4.14921376 8.19712045]
5      成分 1: [5.91278011 5.97285756]
6      成分 2: [1.9522406  2.00239555]
7
8  协方差矩阵:
9      成分 0:
10     [[ 0.62674337 -0.03586796]
11     [-0.03586796  0.75217523]]
12     成分 1:
13     [[ 1.20828202 -0.30783855]
14     [-0.30783855  0.94087822]]
15     成分 2:
16     [[0.84119349 0.22985615]
17     [0.22985615 0.91581628]]
```

对数虽然随着迭代次数的变化如下



用训练处的 GMM 模型进行预测, 可视化预测结果如下:

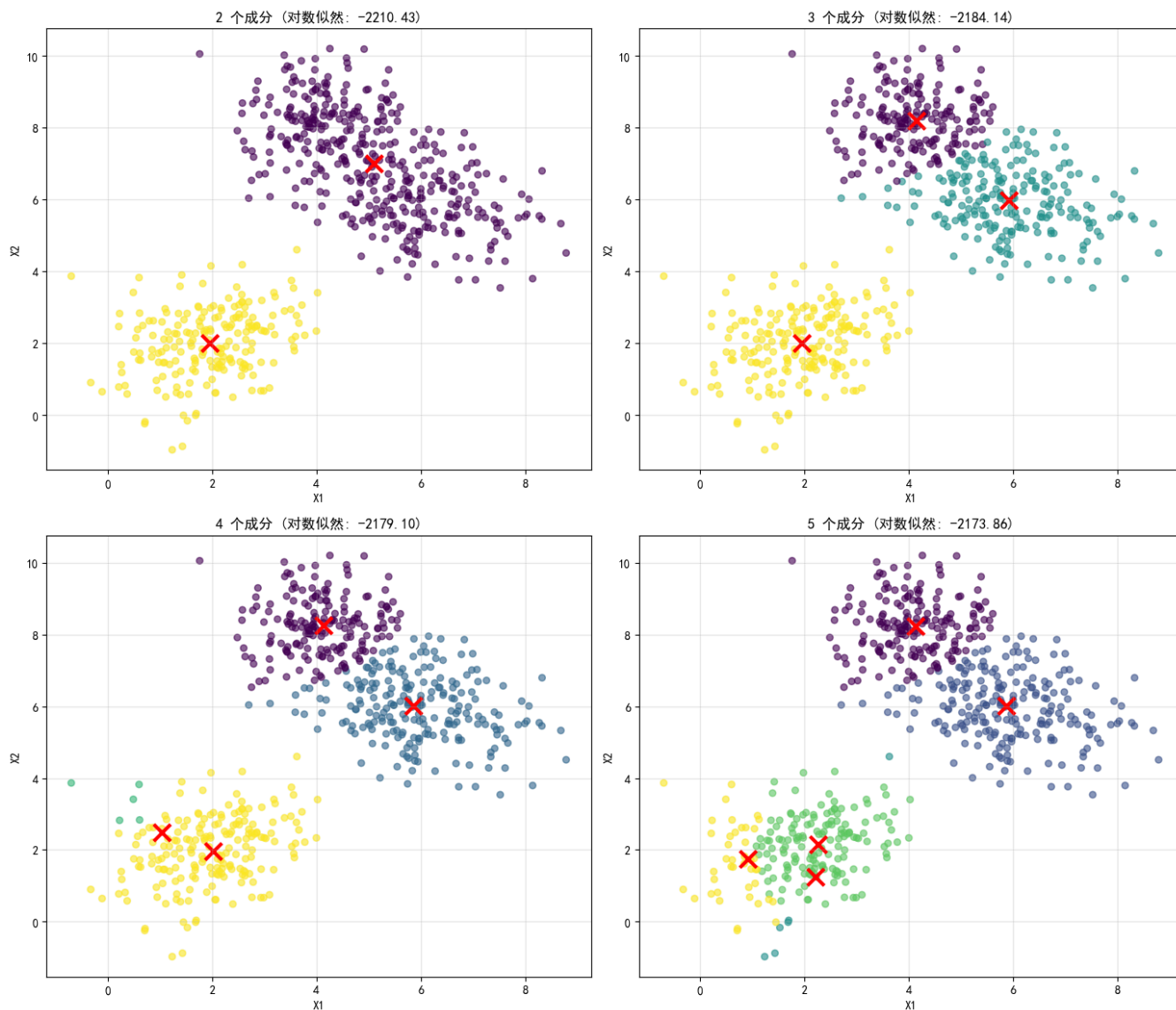


可以发现，学习到三个高斯分布，评估指标如下：

```
1 调整兰德指数 (ARI): 0.7985
2 标准化互信息 (NMI): 0.7831
3
4 最终对数似然: -2184.1402
5
6 各成分的统计信息:
7
8 成分 0:
9   样本数: 186
10  权重: 0.3055
11  均值: [4.14921376 8.19712045]
12  协方差矩阵的行列式: 0.470134
13
14 成分 1:
15  样本数: 215
16  权重: 0.3636
17  均值: [5.91278011 5.97285756]
18  协方差矩阵的行列式: 1.042082
19
20 成分 2:
21  样本数: 199
22  权重: 0.3309
23  均值: [1.9522406 2.00239555]
24  协方差矩阵的行列式: 0.717545
```

比较不同成分数量的效果，





不同成分数量的对数似然: 在第 21 次迭代后收敛

- 2 个成分: -2210.4259
- 3 个成分: -2184.1402
- 4 个成分: -2179.0959
- 5 个成分: -2173.8606

## T3

### 2、Node2vec

数据集: Node2vec\_Dataset.csv。

任务描述: 利用 Node2vec 计算每个节点的 embedding 值。

要求输出: 1) 每个节点的 embedding 值列表 (csv 文件); 2) 随机挑选 10 个 node pair, 对比他们在 embedding 上的相似度和在 betweenness centrality 上的相似度 (使用 Jaccard similarity)。

加载数据集, 构建图。数据集形状为 (27806, 2)

图基本信息如下:

```
1 | 节点数: 7624
2 | 边数: 27806
```

并且图也是连通图

```
1 class Node2Vec:
2     """
3     Node2vec算法实现
4     使用随机游走生成节点序列, 然后使用word2Vec学习节点嵌入
5     """
6
7     def __init__(self, graph, walk_length=80, num_walks=10, p=1, q=1):
8         self.graph = graph
9         self.walk_length = walk_length
10        self.num_walks = num_walks
11        self.p = p
12        self.q = q
13        self.walks = []
14
15    def _get_alias_edge(self, src, dst):
16        """获取边的alias采样表(用于biased random walk)"""
17        unnormalized_probs = []
18        for dst_nbr in sorted(self.graph.neighbors(dst)):
19            if dst_nbr == src:
20                unnormalized_probs.append(1.0 / self.p)
21            elif self.graph.has_edge(dst_nbr, src):
22                unnormalized_probs.append(1.0)
23            else:
24                unnormalized_probs.append(1.0 / self.q)
25
26        norm_const = sum(unnormalized_probs)
27        normalized_probs = [float(u_prob) / norm_const for u_prob in
unnormalized_probs]
28
29        return self._alias_setup(normalized_probs)
30
31    def _alias_setup(self, probs):
32        """设置alias采样表"""
33        K = len(probs)
34        q = np.zeros(K)
35        J = np.zeros(K, dtype=np.int32)
36
37        smaller = []
38        larger = []
39        for kk, prob in enumerate(probs):
40            q[kk] = K * prob
41            if q[kk] < 1.0:
42                smaller.append(kk)
43            else:
44                larger.append(kk)
45
46        while len(smaller) > 0 and len(larger) > 0:
```

```

47         small = smaller.pop()
48         large = larger.pop()
49
50         J[small] = large
51         q[large] = q[large] - (1.0 - q[small])
52
53         if q[large] < 1.0:
54             smaller.append(large)
55         else:
56             larger.append(large)
57
58     return J, q
59
60 def _alias_draw(self, J, q):
61     """从alias表中采样"""
62     K = len(J)
63     kk = int(np.floor(np.random.rand() * K))
64     if np.random.rand() < q[kk]:
65         return kk
66     else:
67         return J[kk]
68
69 def node2vec_walk(self, start_node):
70     """从起始节点开始biased random walk"""
71     walk = [start_node]
72
73     while len(walk) < self.walk_length:
74         cur = walk[-1]
75         cur_nbrs = sorted(self.graph.neighbors(cur))
76         if len(cur_nbrs) > 0:
77             if len(walk) == 1:
78                 walk.append(cur_nbrs[random.randint(0, len(cur_nbrs) - 1)])
79             else:
80                 prev = walk[-2]
81                 J, q = self._get_alias_edge(prev, cur)
82                 next_node = cur_nbrs[self._alias_draw(J, q)]
83                 walk.append(next_node)
84         else:
85             break
86
87     return [str(node) for node in walk]
88
89 def generate_walks(self):
90     """为所有节点生成随机游走序列"""
91     print("生成随机游走序列...")
92     nodes = list(self.graph.nodes())
93
94     for walk_iter in range(self.num_walks):
95         print(f"  游走轮次 {walk_iter + 1}/{self.num_walks}")
96         random.shuffle(nodes)
97         for node in nodes:
98             walk = self.node2vec_walk(node)
99             self.walks.append(walk)

```

```

100
101     print(f"生成了 {len(self.walks)} 条游走序列")
102     return self.walks

```

训练 word2vec模型，并随机挑选10个 node 对比相似度

```

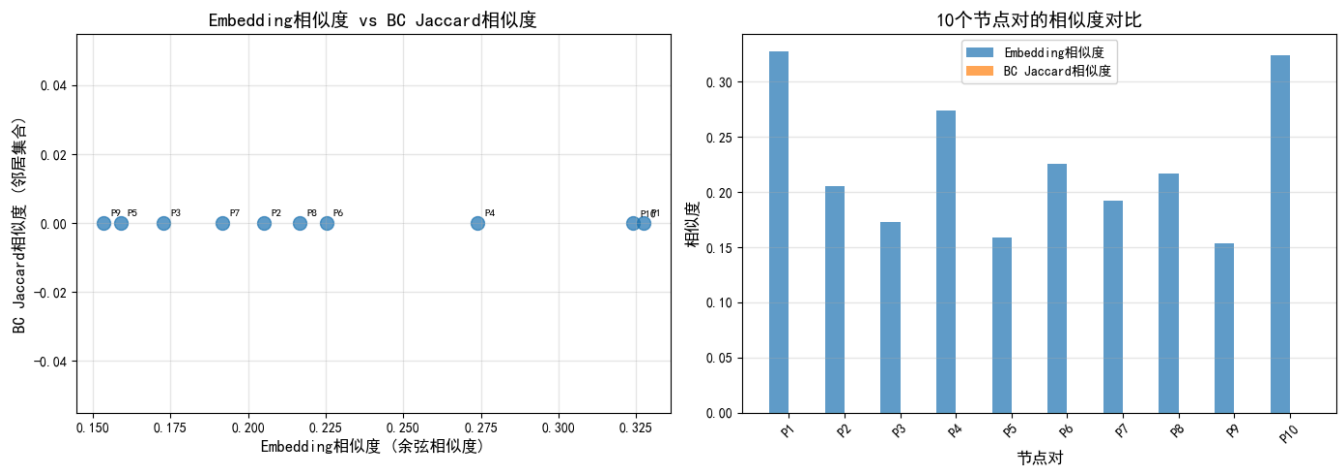
1  随机选择的10个节点对：
2      节点对 1: (6329, 4426)
3      节点对 2: (20, 5258)
4      节点对 3: (5319, 2524)
5      节点对 4: (4279, 2387)
6      节点对 5: (2423, 5644)
7      节点对 6: (4241, 2461)
8      节点对 7: (4812, 4258)
9      节点对 8: (1327, 2769)
10     节点对 9: (792, 3418)
11     节点对 10: (7287, 1149)

```

```

1  节点对相似度对比结果：
2      node_pair  embedding_similarity  bc_jaccard_similarity  betweenness_node1
   betweenness_node2
3      (6329, 4426)          0.327357          0.0          0.000055
   0.003086
4      (20, 5258)           0.205177          0.0          0.000603
   0.000120
5      (5319, 2524)          0.172813          0.0          0.002025
   0.000198
6      (4279, 2387)          0.273827          0.0          0.000027
   0.000762
7      (2423, 5644)          0.158943          0.0          0.000000
   0.000026
8      (4241, 2461)          0.225201          0.0          0.000381
   0.000000
9      (4812, 4258)          0.191887          0.0          0.000010
   0.000000
10     (1327, 2769)          0.216749          0.0          0.000049
   0.000019
11     (792, 3418)           0.153511          0.0          0.000750
   0.002815
12     (7287, 1149)          0.323942          0.0          0.000516
   0.000164

```



## T4

### 3、Clustering

数据集：使用 Make\_blobs 生成数据不少于 1000 个 data points，以 3-5 个 cluster 为宜。

[https://scikit-](https://scikit-learn.org/dev/modules/generated/sklearn.datasets.make_blobs.html#sklearn.datasets.make_blobs)

[learn.org/dev/modules/generated/sklearn.datasets.make\\_blobs.html#sklearn.datasets.make\\_blobs](https://scikit-learn.org/dev/modules/generated/sklearn.datasets.make_blobs.html#sklearn.datasets.make_blobs)

任务描述：利用 DBSCAN 算法计算数据的聚类

要求输出：原始数据 plot 的图像和聚类后的结果。尝试不少于三组  $\epsilon$ ,  $minPts$  的参数组合。

生成 1000 个 data points

```

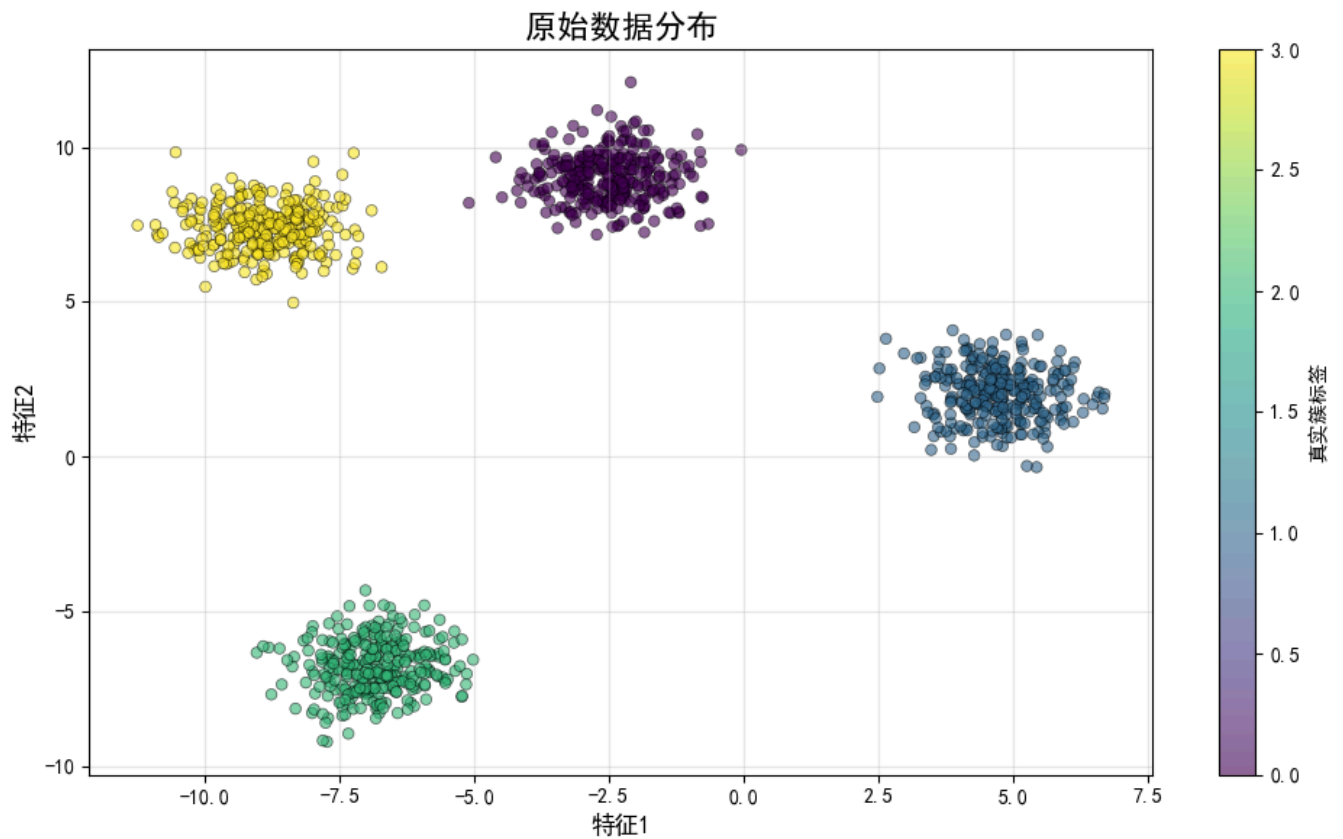
1  np.random.seed(42)
2
3  # 生成4个cluster的数据，总共1200个点
4  x, y_true = make_blobs(
5      n_samples=1200,
6      centers=4,
7      n_features=2,
8      cluster_std=0.8,
9      random_state=42
10 )

```

```

1  数据形状：(1200, 2)
2  真实簇数：4
3  每个簇的样本数：
4      簇 0：300 个样本
5      簇 1：300 个样本
6      簇 2：300 个样本
7      簇 3：300 个样本

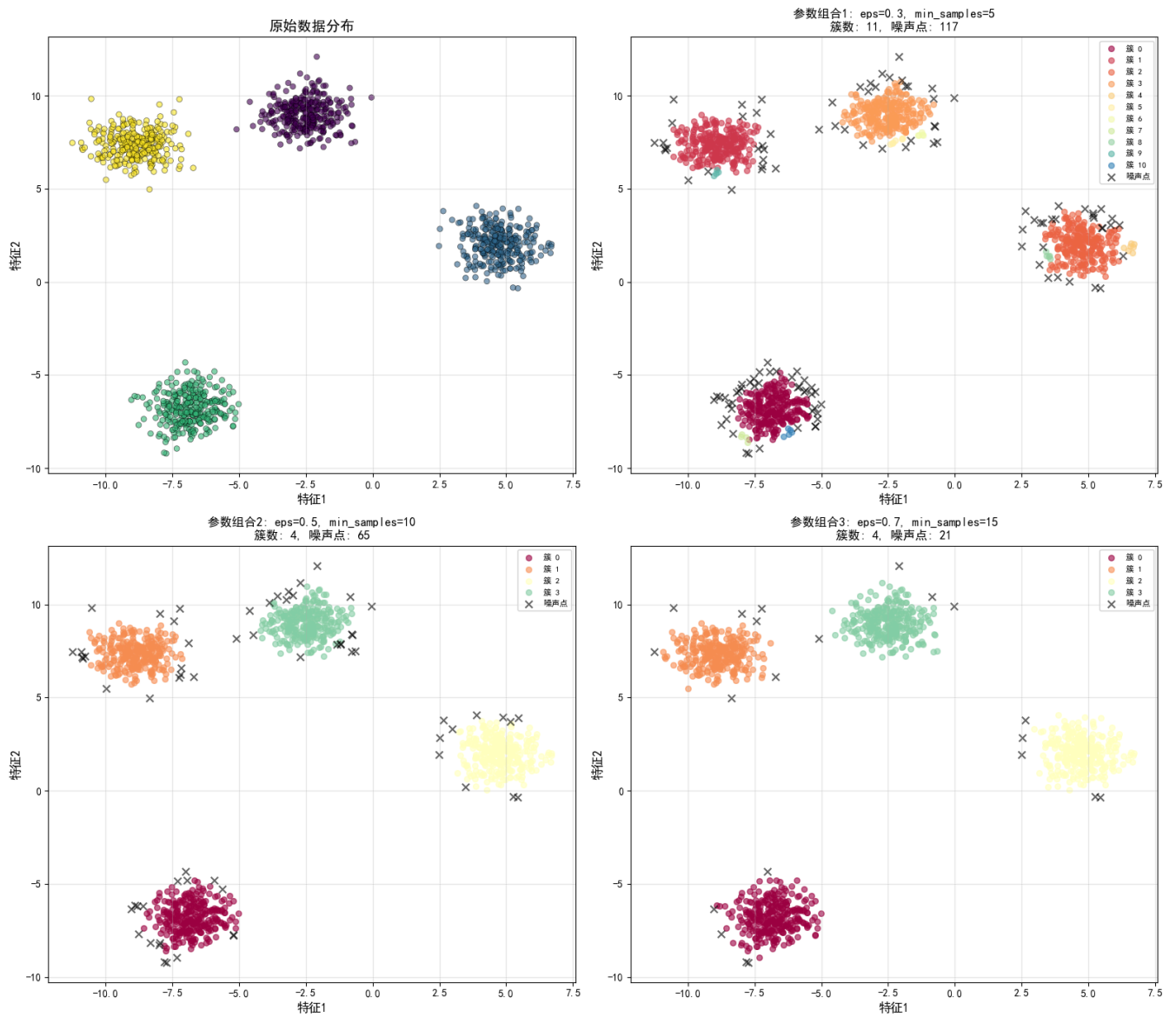
```



使用 DBSCAN 进行聚类，结果如下：

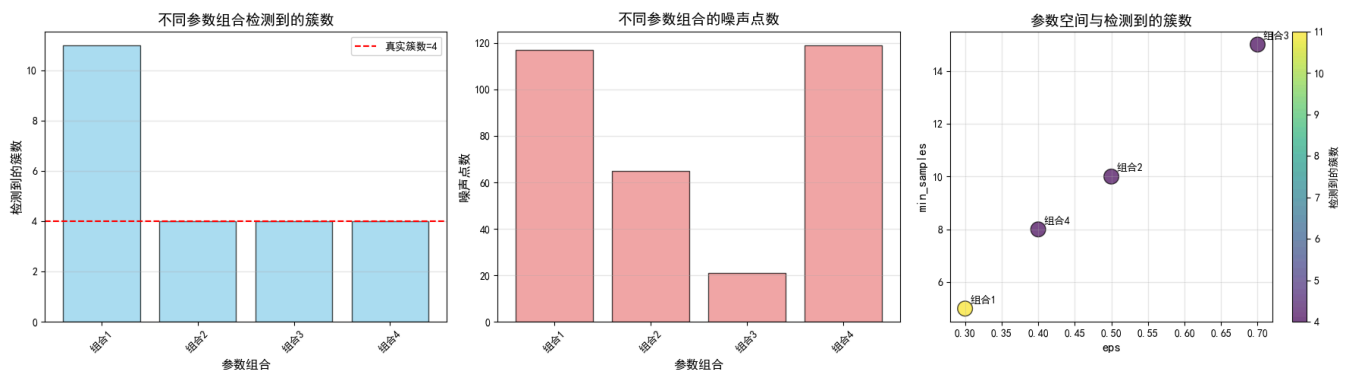
```
1  参数组合1: eps=0.3, min_samples=5:
2      检测到的簇数: 11
3      噪声点数: 117
4      聚类点数: 1083
5
6  参数组合2: eps=0.5, min_samples=10:
7      检测到的簇数: 4
8      噪声点数: 65
9      聚类点数: 1135
10
11 参数组合3: eps=0.7, min_samples=15:
12     检测到的簇数: 4
13     噪声点数: 21
14     聚类点数: 1179
15
16 参数组合4: eps=0.4, min_samples=8:
17     检测到的簇数: 4
18     噪声点数: 119
19     聚类点数: 1081
```

可视化结果如下：



## 使用不同的参数组合

1	参数组合对比:						
2		参数组合	eps	min_samples	检测到的簇数	噪声点数	聚类点数
3	参数组合1:	$\text{eps}=0.3$ , $\text{min\_samples}=5$	0.3	5	11	117	1083
4	参数组合2:	$\text{eps}=0.5$ , $\text{min\_samples}=10$	0.5	10	4	65	1135
5	参数组合3:	$\text{eps}=0.7$ , $\text{min\_samples}=15$	0.7	15	4	21	1179
6	参数组合4:	$\text{eps}=0.4$ , $\text{min\_samples}=8$	0.4	8	4	119	1081



```
1  参数组合1: eps=0.3, min_samples=5:
2      eps = 0.3, min_samples = 5
3      结果: 检测到 11 个簇, 117 个噪声点
4      各簇大小: {0: 248, 1: 274, 2: 261, 3: 264, 4: 7, 5: 6, 6: 5, 7: 4, 8: 5, 9: 4, 10: 5}
5      聚类比例: 90.25%
6
7  参数组合2: eps=0.5, min_samples=10:
8      eps = 0.5, min_samples = 10
9      结果: 检测到 4 个簇, 65 个噪声点
10     各簇大小: {0: 282, 1: 284, 2: 289, 3: 280}
11     聚类比例: 94.58%
12
13 参数组合3: eps=0.7, min_samples=15:
14     eps = 0.7, min_samples = 15
15     结果: 检测到 4 个簇, 21 个噪声点
16     各簇大小: {0: 295, 1: 293, 2: 295, 3: 296}
17     聚类比例: 98.25%
18
19 参数组合4: eps=0.4, min_samples=8:
20     eps = 0.4, min_samples = 8
21     结果: 检测到 4 个簇, 119 个噪声点
22     各簇大小: {0: 265, 1: 278, 2: 271, 3: 267}
```

- eps参数控制邻域半径, 值越大, 越容易形成更大的簇
- min\_samples参数控制形成核心点的最小样本数, 值越大, 对噪声越敏感