DBSCAN实验报告

0. 引言

DBSCAN是一种密度聚类算法，用于发现具有相似密度的数据点组成的簇，并能识别噪声点。它不需要预先指定簇的数量，能够自动适应数据的分布形状。DBSCAN的基本流程如下：

1. 选择一个未被访问的数据点P。
2. 找出P的邻域内的所有数据点。
3. 如果邻域内的数据点数大于等于指定的阈值，则将P及其邻域内的数据点作为一个簇。
4. 对于邻域内的每个未被访问的数据点，重复步骤2和步骤3，将这些点添加到簇中。
5. 将没有足够邻域内数据点的点标记为边界点，不分配到任何簇。
6. 重复步骤1至步骤5，直到所有数据点都被访问过。

DBSCAN算法的关键参数是邻域半径和邻域内最小数据点数。它对于发现任意形状的簇和处理噪声点具有优势。但在高维数据和不同密度之间的数据集上可能面临挑战。改进和扩展算法也被提出来解决这些问题。

1. 算法设计

1.1 时空复杂度优化

1. **KDTree**: 使用`KDTree`数据结构来存储数据点，这是一种可以高效处理多维空间搜索问题的数据结构。在这里，它被用来快速找出给定点的邻居点，这大大减少了搜索时间复杂度。
2. **批量查询**: `query_ball_point`方法允许一次性对所有点进行邻居搜索，而不是对每个点单独搜索。这样可以减少重复的搜索操作，从而降低时间复杂度。
3. **标记噪声**: 通过检查每个点的邻居数量是否小于`min_samples`，直接将其标记为噪声（`self.labels_[i] = 0`），避免了对这些点的进一步处理，节省了空间和时间资源。
4. **集群ID管理**: 使用`cluster_id`变量来跟踪当前的集群编号，这样可以避免使用额外的数据结构来存储集群信息，从而节省空间。
5. **核心点索引**: `self.core_samples_indices_`列表存储核心点的索引，这样可以在不增加额外空间的情况下快速访问核心点。
6. **并行处理**: 使用`ThreadPoolExecutor`来并行处理点，这样可以利用多核处理器的能力，加快处理速度，降低时间复杂度。
7. **集合操作**: 使用集合（`set`）来处理种子点，因为集合在Python中是基于哈希表实现的，这意味着添加和删除操作的平均时间复杂度是O(1)，这比列表操作要高效得多。

1.2 算法详细介绍

```python
class DBSCAN:
    def __init__(self, eps=0.5, min_samples=5):
        self.eps = eps
        self.min_samples = min_samples

    def fit(self, X):
        self.core_samples_indices_ = []
        self.labels_ = -np.ones(len(X), dtype=int)
        cluster_id = 0
        tree = KDTree(X)
        all_neighbors = tree.query_ball_point(X, self.eps)  # 批量查询

        # 定义一个处理每个点的函数
        def process_point(i):
            if self.labels_[i] != -1:
                return

            neighbors = all_neighbors[i]
            if len(neighbors) < self.min_samples:
                self.labels_[i] = 0  # Mark as noise
                return

            nonlocal cluster_id
            cluster_id += 1  # Start a new cluster
            self.labels_[i] = cluster_id
            self.core_samples_indices_.append(i)

            seeds = set(neighbors)
            seeds.remove(i)

            while seeds:
                j = seeds.pop()
                if self.labels_[j] == 0:
                    self.labels_[j] = cluster_id  # Change noise to border point
                if self.labels_[j] != -1:
                    continue  # Already processed

                self.labels_[j] = cluster_id
                neighbors = all_neighbors[j]
                if len(neighbors) >= self.min_samples:
                    seeds.update(neighbors)

        # 使用线程池并行处理每个点
        with ThreadPoolExecutor() as executor:
            executor.map(process_point, range(len(X)))

        return self
```

核心算法封装在一个类`DBSCAN`中，下面是对类方法的介绍：

1. **初始化 (`__init__` 方法)**: 这个方法初始化了两个重要的参数：
   - `eps`（epsilon）: 这是用来确定邻域大小的半径。在这个半径内的点被认为是相邻的。
   - `min_samples`: 这是一个点要成为核心点所需的最小邻居数目。
2. **拟合 (`fit` 方法)**: 这个方法是算法的核心，它接收一个数据集`X`作为输入，并执行以下步骤：
   - 初始化核心样本索引列表`self.core_samples_indices_`和标签数组`self.labels_`。
   - 创建一个`KDTree`实例来加速邻近点的搜索。
   - 使用`query_ball_point`方法批量查询每个点的邻居。
3. **处理每个点 (`process_point` 函数)**: 这是一个内部函数，用于处理单个数据点：
   - 如果一个点已经被处理过（即`self.labels_[i]`不是-1），则跳过它。
   - 如果一个点的邻居少于`min_samples`，则将其标记为噪声（标签为0）。
   - 否则，为该点创建一个新的簇，并将其邻居添加到种子集合中以进一步探索。
4. **并行处理**: 使用`ThreadPoolExecutor`来并行执行`process_point`函数，这样可以同时处理多个点，提高算法的效率。

5. **更新簇和边界点**: 在处理种子集合时，算法会不断地将新的邻居点加入当前簇，并将噪声点转变为边界点。

1.3 时空复杂度分析

时间复杂度:

1. 构建`KDTree`的时间复杂度通常是`O(nlogn)`，其中`n`是数据点的数量。
2. `query_ball_point`方法的时间复杂度取决于数据点的分布，但在平均情况下，对于每个点的查询时间复杂度是`O(logn)`，因此对所有点进行查询的总时间复杂度是`O(nlogn)`。
3. `process_point`函数中的循环可能会多次执行，但每个点最多被处理一次，因此这部分的时间复杂度是`O(n)`。
4. 使用线程池并行处理可以显著减少实际运行时间，但这不会改变算法的理论时间复杂度。

综上所述，DBSCAN算法的平均时间复杂度大约是`O(nlogn)`，但这可能会因为数据的特殊分布而变得更好或更坏。

空间复杂度:

1. `KDTree`的空间复杂度是`O(n)`。
2. `labels_`数组和`core_samples_indices_`列表的空间复杂度也是`O(n)`。
3. `all_neighbors`列表包含每个点的邻居索引，其空间复杂度是`O(n)`。

因此，DBSCAN算法的总空间复杂度大约是`O(n)`。

2. 实验设置

2.1 数据集选择

进行DBSCAN算法测试时，应该考虑以下几个因素：

1. **数据集大小**：选择一个适中大小的数据集可以帮助您快速运行算法并观察结果。太大的数据集可能会导致长时间的运行，而太小的数据集可能无法充分展示算法的性能。
2. **维度**：由于DBSCAN对高维数据的性能可能下降，选择一个低维（例如2维或3维）的数据集可以更直观地观察聚类结果。
3. **数据分布**：选择具有明显聚类结构的数据集可以帮助您验证DBSCAN实现的有效性。数据集中的聚类应该是密集的，且聚类之间有明显的分隔。
4. **噪声数据**：DBSCAN能够处理噪声数据，因此选择一个包含一定噪声的数据集可以测试算法在实际应用中的鲁棒性。

基于这些标准，考虑使用***Iris数据集***。这个数据集包含150个样本，每个样本有4个特征，分为3个类别。这个数据集的优点是：

- **适中的大小**：150个样本足以展示DBSCAN的聚类能力，同时计算量不会太大。
- **低维度**：虽然有4个特征，但您可以选择其中的2个或3个进行可视化，以便直观地观察聚类结果。
- **明显的聚类结构**：Iris数据集中的类别在特征空间中有较好的分离，适合测试聚类算法。
- **包含噪声**：虽然Iris数据集相对较干净，但在某些特征组合下，类别之间的边界并不是非常清晰，这可以作为噪声数据的一个代表。

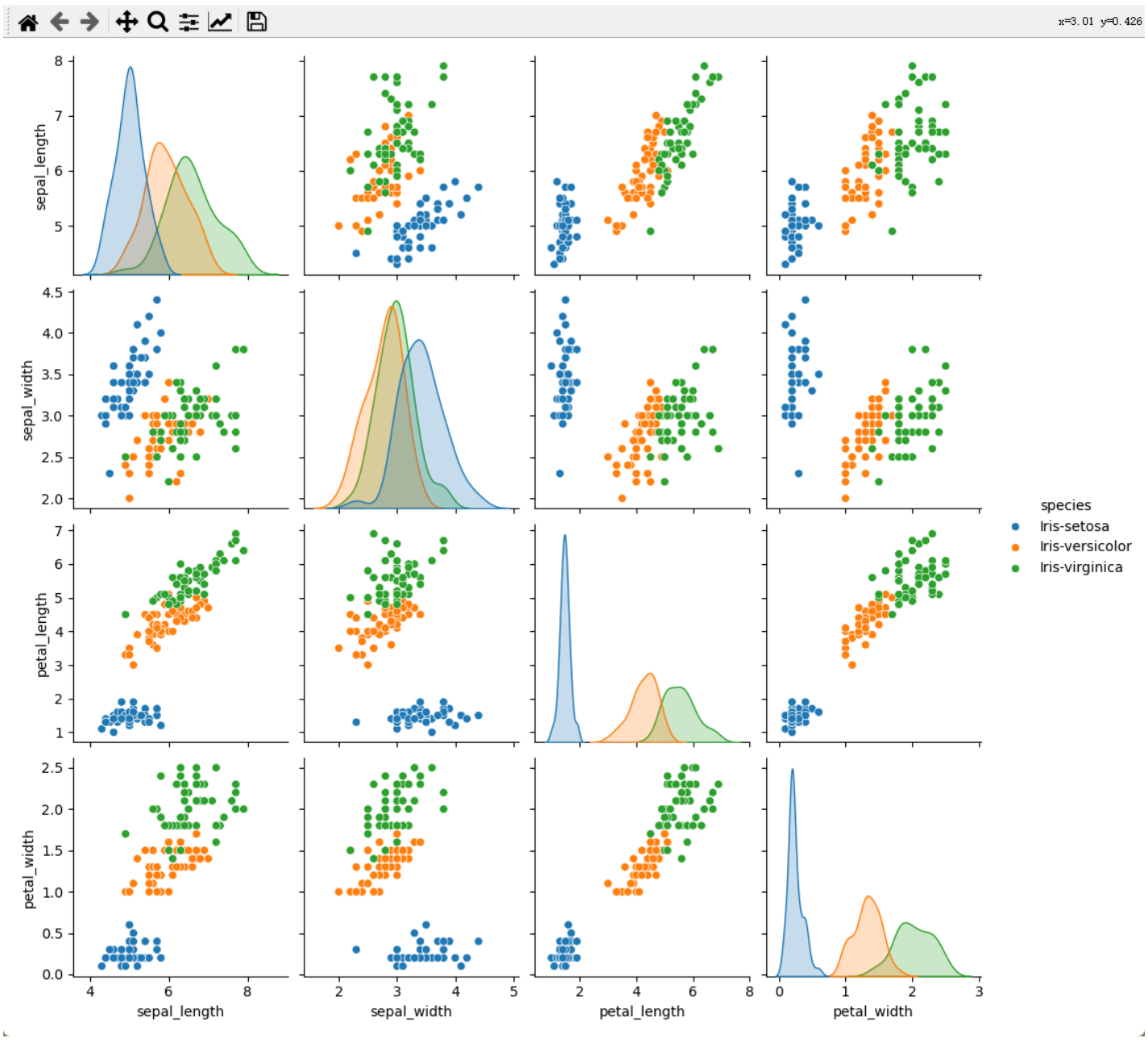
使用Iris数据集进行测试，可以观察DBSCAN如何根据密度将数据点分配到不同的聚类中，以及如何标记噪声点。

2.2 实验流程和聚类指标

1. 可视化观察iris数据集的数据点聚类分布情况（散点图矩阵），选择合适的两个维度来做DBSCAN，以便于DBSCAN的可视化并且展示DBSCAN聚类算法的特点。
2. 对上一步选择的两个维度选择合适的eps（邻域半径）和minPts（最小点数）进行聚类，然后对聚类可视化，同时输出**聚类指标（轮廓系数，DB指数）**和运行时间等。
3. 调整参数进行多次（使用**ablation study 消融实验**），观察结果。
4. 整合结果观察，进行分析并总结DBSCAN聚类算法的特点和自己的观点。

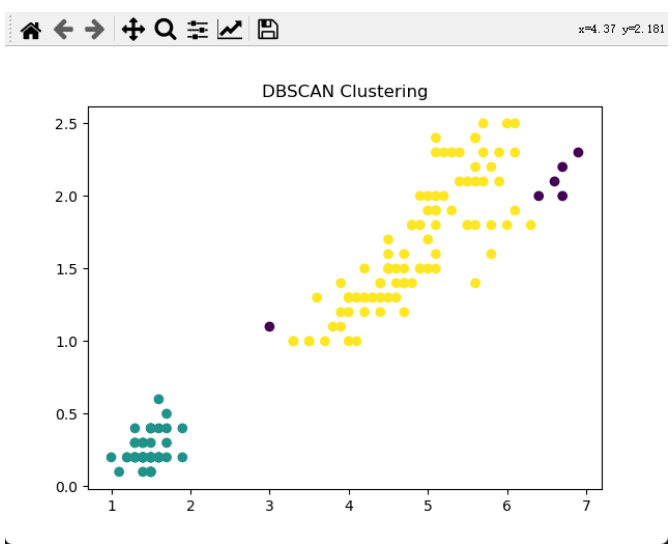
3. 实验结果

1. 可视化观察iris数据集：

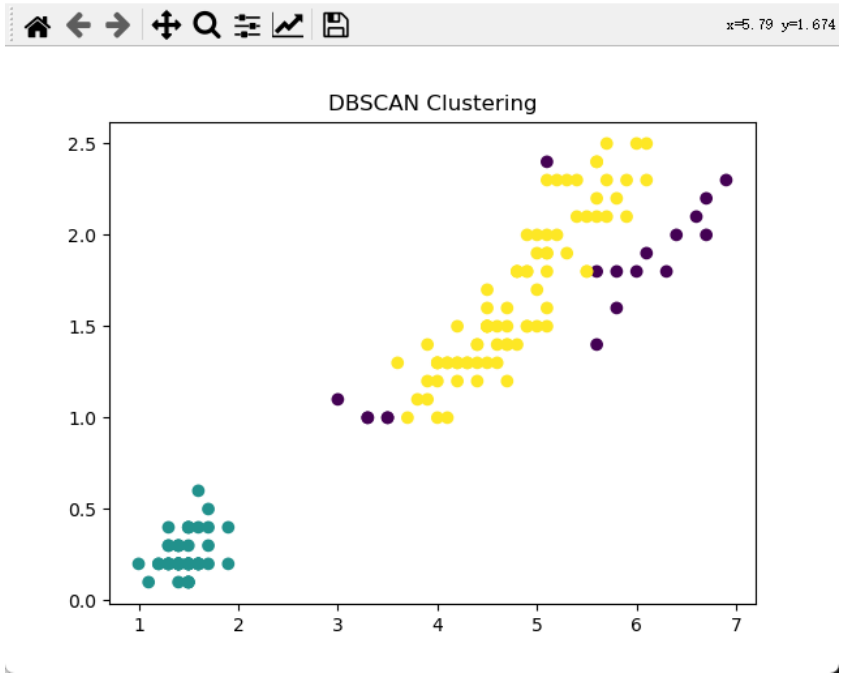


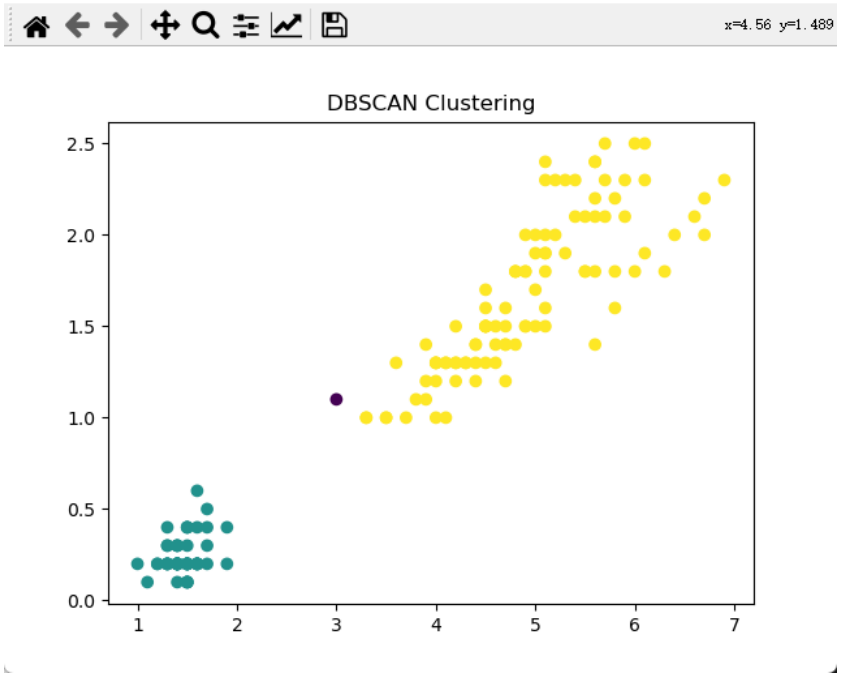
我们选择特征`petal_length`和`petal_width`来进行我们的实验，因为肉眼观察上面的图我们能发现这两个特征的图具有明显的密度聚类特点和多离散点分布，有利于显示DBSCAN聚类算法的特点。

2. 我们首先按照肉眼选择合适设定参数为`eps=0.3`,`min_samples=5`，得到结果如下图，运行时间为`0.0148s`，轮廓系数为`0.5910`，DB指数为`1.0310`。

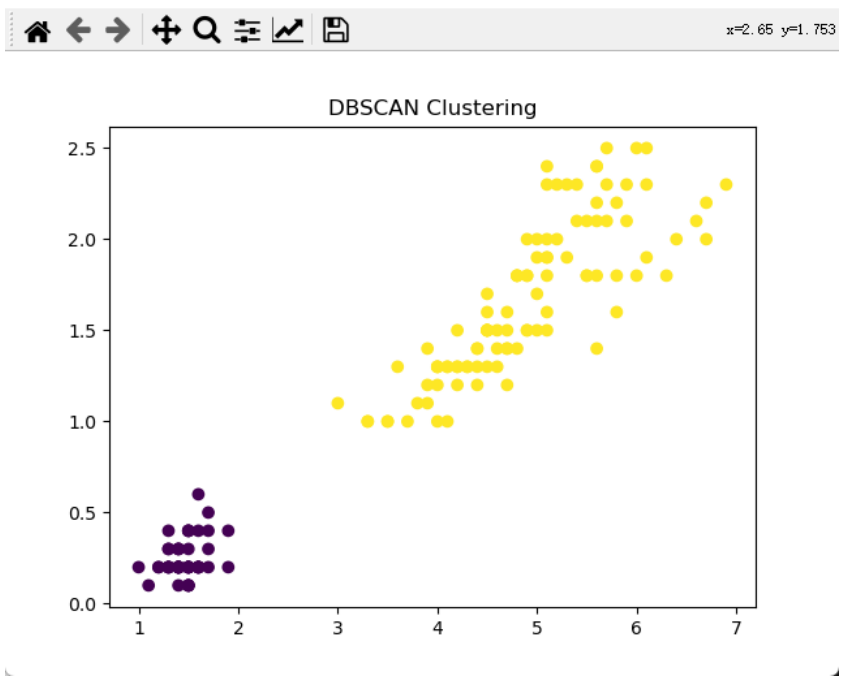


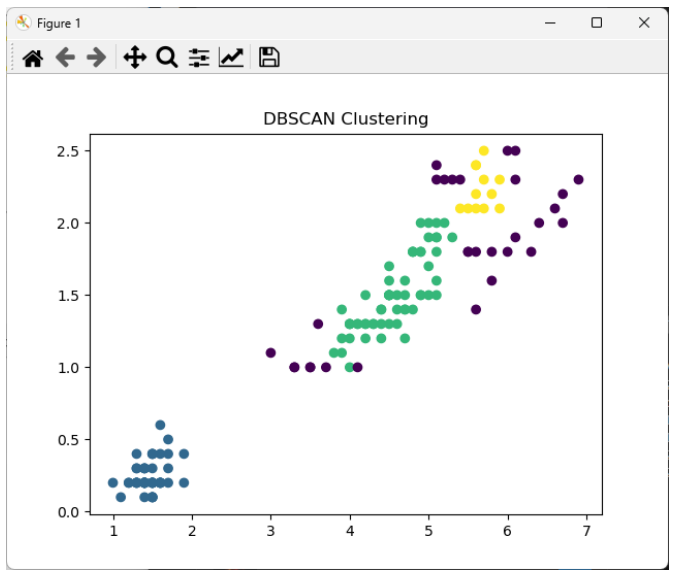
3. 然后我们保持`eps=0.3`不变，分别提高降低`min_samples`到`7`和`3`，运行时间分别为`0.0135s`和`0.0161s`，轮廓系数分别为`0.5398`和`0.5344`，DB指数分别为`1.6172`和`0.3426`。





4. 然后我们保持 `min_samples=5` 不变，分别提高降低 `eps` 到 `0.4` 和 `0.2`，运行时间分别为 `0.0156s` 和 `0.0133` s，轮廓系数分别为 `0.7667` 和 `0.4592`，DB指数分别为 `0.2575` 和 `1.6638` 。





DBSCAN参数实验结果表

参数	eps	min_samples	运行时间（s）	轮廓系数	DB指数
初始设定	0.3	5	0.0148	0.5910	1.0310
min_samples=7	0.3	7	0.0135	0.5398	1.6172
min_samples=3	0.3	3	0.0161	0.5344	0.3426
eps=0.4	0.4	5	0.0156	0.7667	0.2575
eps=0.2	0.2	5	0.0133	0.4592	1.6638

4. 实验分析

根据以上**ablation study 消融实验**的结果，我们可以看出：

min_samples参数的大小与运行时间的大小呈负相关关系

`min_samples` 参数定义了一个点要成为核心点（即一个密度可达的点）所需的邻域内的最小样本数。当你增加 `min_samples` 的值时，算法将只在更高密度的区域中寻找核心点，这意味着它会忽略更多的边缘点和噪声点，从而减少了需要处理的点的数量。的大小与运行时间的大小呈负相关关系

eps参数的大小与运行时间的大小呈正相关关系

`eps` 参数定义了点与点之间的最大距离，即一个点的邻域半径。当增加 `eps` 的值时，每个点的邻域内包含的点数也会增加。这意味着算法需要处理更多的点来确定核心点和边界点，从而增加了计算量和运行时间。

增加 `eps` 参数:

- 聚簇的数量可能会减少，因为更多的点会被包含在现有的聚簇中。

- 聚簇的大小会增加，因为每个点的邻域范围扩大，更多的点被视为核心点的一部分。
- 离群点的数量可能会减少，因为一些原本被视为离群点的点现在可能被包含在聚簇中。

减少 `eps` 参数：

- 聚簇的数量可能会增加，因为点与点之间的连接减少，导致原本属于同一个聚簇的点可能被分割成多个小聚簇。
- 聚簇的大小会减小，因为邻域范围缩小，少数点满足核心点的条件。
- 离群点的数量可能会增加，因为更多的点无法被归入任何聚簇。

增加 `min_samples` 参数：

- 聚簇的数量可能会减少，因为需要更多的邻近点来形成一个聚簇。
- 聚簇的大小可能会减小，因为只有那些在高密度区域的点才能成为核心点。
- 离群点的数量可能会增加，因为更多的点不满足核心点的条件。

减少 `min_samples` 参数：

- 聚簇的数量可能会增加，因为更少的邻近点就能形成一个聚簇。
- 聚簇的大小可能会增加，因为更容易找到核心点。
- 离群点的数量可能会减少，因为更多的点可以被归入聚簇。

轮廓系数 (Silhouette Coefficient)：

- **轮廓系数**是衡量聚类效果的一个指标，它结合了聚类的紧密度和分离度两个方面。轮廓系数的值介于-1到1之间，值越高表示聚类结果越好。
- 在DBSCAN算法中，轮廓系数可以帮助我们理解不同 `eps` 和 `min_samples` 参数设置下，聚类的形状和大小如何变化。例如，当 `eps=0.4` 时，轮廓系数达到最高（0.7667），这表明在较大的 `eps` 值下，聚类内部的点更加紧密，且聚类之间的分离度也更高，这是一个理想的聚类状态。
- 反之，当 `eps=0.2` 时，轮廓系数最低（0.4592），可能意味着聚类内部的点不够紧密，或者不同聚类之间的分离度不够，这可能是由于 `eps` 值过小，导致聚类过于分散。

DB指数 (Davies-Bouldin Index)：

- **DB指数**是另一个评估聚类质量的指标，它基于聚类内部的紧密度和聚类之间的分离度来计算。DB指数越低，表示聚类质量越高。
- 在DBSCAN算法中，DB指数可以反映不同参数设置下聚类的紧密度和分离度。例如，当 `eps=0.4` 时，DB指数最低（0.2575），这与轮廓系数的高值相对应，表明这是一个高质量的聚类结果。
- 当 `min_samples=7` 时，DB指数最高（1.6172），这可能表明在较高的 `min_samples` 值下，聚类内部的紧密度不足，或者聚类之间的分离度过低，这可能是因为较高的 `min_samples` 值导致算法只在高密度区域形成聚类，忽略了较多的边缘点。

5. 实验结论和见解

1. `min_samples` **参数对运行时间的影响**：
   - 较大的 `min_samples` 值会导致算法在更高密度的区域寻找核心点，减少了边缘点和噪声点的处理，从而减少运行时间。这一发现对于在大规模数据集上应用DBSCAN算法具有重要意义，因为它提供了一种通过调整 `min_samples` 来优化算法性能的方法。
2. `eps` **参数对运行时间和聚类质量的双重影响**：
   - `eps` 参数的增加会导致每个点的邻域内包含的点数增加，增加了算法的计算量，从而增加运行时间。然而，适当的 `eps` 值能够显著提高聚类质量，如 `eps=0.4` 时的轮廓系数和DB指数所示。
3. **聚类质量评估**：
   - 轮廓系数和DB指数作为聚类质量的评估指标，为我们提供了量化聚类性能的手段。特别是在 `eps=0.4` 的设置下，高轮廓系数和低DB指数表明了优秀的聚类质量，这强调了选择合适 `eps` 值的重要性。
4. **参数选择的重要性**：
   - DBSCAN算法的参数选择对聚类结果有着决定性的影响。合适的 `eps` 和 `min_samples` 参数不仅影响算法的运行效率，还直接关系到聚类结果的质量。
5. **算法应用的策略**：

- 在实际应用中，我们应该根据数据集的特性和聚类任务的需求来调整DBSCAN参数。例如，对于噪声较多的数据集，可以通过增加 `min_samples` 值来提高聚类的鲁棒性。