DBSCAN聚类算法报告

数据集的选择理由

1. **数据集规模适中**：鸢尾花数据集大小适中，包含150个样本，每个样本有4个特征，这使得它既可以体现出数据处理方法的效果，又不会因为数据量过大而导致计算太过复杂或耗时。
2. **特征清晰，具有标签**：鸢尾花数据集的每个样本都有明确的标签，即属于哪一种鸢尾花。这便于在聚类之后可以利用这些标签评估聚类的效果。
3. **适合展示聚类效果**：鸢尾花数据集在特征空间中可以形成较好的簇结构，非常适合用来展示和测试聚类算法，其标准化后的两个维度的可视化效果很好。
4. **普遍性和可获取性**：作为经典数据集，鸢尾花数据集被广泛用于各种机器学习的教学和研究中，大多数人对它都比较熟悉，这有助于理解和评估算法的效果。同时，它在很多库，如`sklearn`中都可以直接获取。

算法设计

代码实现了一个基于密度的聚类算法`DBSCAN`（`Density-Based Spatial Clustering of Applications with Noise`），下面是对算法的分析：

1. **数据加载和预处理**：
   - 使用`Pandas`加载数据，并选择其中的`petal length`和`petal width`两列。
   - 使用`StandardScaler`对数据进行了标准化处理。标准化是一种常见的预处理技术，它通过减去特征的均值并除以标准差，将特征的值缩放到均值为 0，标准差为 1 的标准正态分布。这样做可以使得不同特征的尺度一致，避免某些特征对聚类结果的影响过大。
2. `DBSCAN`**算法实现**：
   - **KD 树构建**：
     - KD 树是一种二叉树数据结构，它被用来加速邻域查询过程。KD 树将数据集分割成多个区域，每个节点代表一个超矩形区域，其子节点对应于该区域被划分后的子区域。
     - 在代码中，通过`KDTree(D)`构建了一个 KD 树，其中`D`是数据集，即经过标准化处理后的特征矩阵。KD 树的构建过程在算法执行前只需执行一次，之后可以用于快速查找每个点的邻域。
   - `DBSCAN`**算法主要逻辑**：
     - 在`` `DBSCAN`_manual`函数中，首先初始化了一些变量，如簇标签`labels`、簇编号`cluster_id`、访问状态`visited`` 等。
     - 然后，通过遍历数据集中的每个点来执行`DBSCAN`算法：
       - 对于每个未被访问的点，首先将其标记为已访问，并找到其邻域内的所有点。
       - 如果邻域内的点数量大于等于`minPts`，则将该点标记为核心点，并扩展以该核心点为中心的簇。
       - 扩展簇的过程是通过不断地将邻域内的点添加到簇中，并标记它们为已访问来实现的。
       - 如果某个邻域内的点不足以形成一个簇（即小于`minPts`），则将其标记为噪声点。
   - **邻域查询**：

- 在`DBSCAN`算法中，邻域查询是指确定每个点的邻域内包含哪些其他点。这是通过定义一个半径范围来实现的，该范围内的所有点都被视为邻居。具体来说，邻域查询用于找到距离某一点不超过ε的所有点，其中ε是`DBSCAN`算法中的一个重要参数，表示点的领域半径。
- 在代码中，邻域查询通过KD树实现，使用了`tree.query_radius([D[p_index]], eps)[0]`这行代码。这行代码中，`tree.query_radius()`函数会返回距离给定点`D[p_index]`在半径为`eps`范围内的所有点的索引。这个函数返回的索引数组表示了所有在给定点邻域内的点的位置。

- **簇的扩展**：
  - 簇的扩展是`DBSCAN`算法的另一个核心步骤，它负责将核心点的邻域内的点添加到同一个簇中，并递归地对新添加的点进行进一步扩展。在`DBSCAN`算法中，当发现一个核心点时，就会启动簇的扩展过程。这个过程是通过遍历核心点的邻域内的所有点来实现的。对于每个邻域内的点，如果它还没有被分配到任何簇中，那么它就会被添加到当前簇中，并且如果它也是一个核心点，那么会对它的邻域进行进一步的扩展。这个过程会一直进行下去，直到没有更多的点可以添加到簇中为止。
  - 在代码中，簇的扩展是通过`expand_cluster`函数来实现的。这个函数通过遍历邻域内的点，并将它们添加到当前簇中，然后递归地对新添加的点进行邻域查询和扩展。这个过程会一直进行下去，直到邻域内没有更多的点可以被添加到簇中为止。

3. **参数选择**：
   - **领域半径`eps`**：
     - `eps`是DBSCAN算法中一个非常关键的参数，它决定了一个点的邻域的范围。
     - 选择合适的`eps`值对于聚类的效果至关重要。如果`eps`设置得太小，会导致大部分点被标记为噪声点，而聚类的效果不佳；如果`eps`设置得太大，可能会将不同的簇合并到一起，导致聚类结果模糊不清。
     - 通常，可以通过可视化方法（如领域图或核心点图）来调整`eps`的值。另一种方法是使用基于密度的方法来确定`eps`的值，例如通过寻找距离点密度下降的“肘部”，从而确定合适的`eps`值。
   - **最小邻居数`minPts`**：
     - `minPts`是DBSCAN算法中用于确定核心点的参数，它定义了一个点的邻域中至少包含的点的数量。选择合适的`minPts`值需要考虑数据的密度分布。对于高密度数据集，可以选择较大的`minPts`值；而对于低密度数据集，可能需要选择较小的`minPts`值。`minPts`参数的选择也可以通过可视化方法来进行调整，例如通过观察核心点的分布情况来确定合适的`minPts`值。
     - 通常，建议从较小的值开始，然后逐渐增加`minPts`的值，直到得到满意的聚类结果为止。同时，也可以结合对`eps`参数的调整来进一步优化聚类效果。
   - **选择方式**：
     - 在选取合适的参数组合时，我采用网格搜索法，遍历每一种组合最终得到最优参数
     - ```
       # 循环遍历参数组合
       for eps in eps_range:
           for minPts in minPts_range:
               clusters = dbscan_manual(data_scaled, eps, minPts)
               if len(set(clusters)) > 1:  # 确保有多于一个聚类（避免全部为噪声）
                   ari = adjusted_rand_score(labels_true, clusters)
                   if ari > best_ari:
                       best_ari = ari
       ```
       ```

```
                best_params['eps'] = eps
                best_params['minPts'] = minPts

# 输出最佳参数
print("最佳参数组合:", best_params)
print("最佳 ARI 分数:", best_ari)
```

- 结果：

```
C:\Users\10728\AppData\Local\Programs\Python
最佳参数组合: {'eps': 0.4, 'minPts': 35}
最佳 ARI 分数: 0.8177533513813235
```

4. **评估聚类效果**：
   - 评估聚类效果是在完成聚类算法后，对聚类结果进行量化分析和比较的过程。常用的评估指标有纯度（`Purity`）、标准化互信息（`Normalized Mutual Information，NMI`）和调整兰德指数（`Adjusted Rand Index，ARI`）。下面对这些评估指标进行详细说明：
     - 纯度（`Purity`）：
       - 纯度是一种简单而直观的聚类评估指标，用于衡量每个聚类中包含的样本是否都属于同一类别。纯度越高，表示聚类结果越好。
       - 纯度的计算方式是对于每个聚类，选择其中最常见的真实类别作为该聚类的标签，然后将所有正确分类的样本数相加，再除以总样本数。数学公式如下：

$$textPurity = \frac{1}{N}\sum_{k}\max_{j}|C_k \cap T_j|$$

         其中，N是样本总数，C_k是第k个聚类，T_j是第j个真实类别。
     - 标准化互信息（`NMI`）：
       - 标准化互信息是一种用于衡量两个分布之间相似性的指标，常用于衡量聚类结果与真实标签之间的一致性。NMI 越接近 1，表示聚类结果与真实标签的一致性越高。
       - NMI 的计算方式涉及到信息论中的熵和互信息的概念，它对聚类结果和真实标签的分布进行了统计。
       - NMI 的取值范围在 0 到 1 之间，当取值为 0 时表示聚类结果与真实标签之间没有任何关联，取值为 1 时表示两者完全一致。
     - 调整兰德指数（ARI）：
       - 调整兰德指数是一种用于衡量两个数据分布的相似性的指标，通常用于评估聚类算法的性能。
       - ARI 考虑了所有样本对之间的相似性，包括聚类结果和真实标签之间的匹配情况以及不匹配情况。
       - ARI 的取值范围在 -1 到 1 之间，当取值为 1 时表示聚类结果与真实标签完全一致，取值为 0 时表示随机聚类结果，取值为 -1 时表示聚类结果与真实标签完全相反。
   - 对取值的要求：
     - 纯度、NMI 和 ARI 的取值越高越好，它们都是用于衡量聚类结果的质量的指标。
     - 对于纯度来说，理想情况下应接近于 1，表示每个聚类都是纯净的，包含相同类别的样本。
     - 对于 NMI 和 ARI 来说，理想情况下应接近于 1，表示聚类结果与真实标签之间的一致性很高。

综上所述，纯度、NMI 和 ARI 是评估聚类效果常用的指标，它们能够提供对聚类结果质量的不同角度的评价，帮助分析和比较不同聚类算法的性能。

代码实现了一个完整的 `DBSCAN` 聚类过程，包括参数选择、聚类执行、结果可视化和聚类效果评估。通过这些步骤，可以对数据集进行密度聚类，并评估聚类结果的质量。

复杂度分析

- **时间复杂度**：每一次`region_query`的操作复杂度是$O(logN)$，它被调用了N次，所以这部分复杂度为$O(NlogN)$。`expand_cluster`中的while循环，在最坏情况下，会运行N^2次（所有数据点都在一个簇中）。所以，时间复杂性为$O(N^2logN)$。
- **空间复杂度**：你在代码中使用了一些存储结构，包括数据集D，KD树，标签列表，访问记录，以及局部变量。但是，所有这些内存占用量都是和输入规模线性相关的。因此，DBSCAN的空间复杂性为$O(N)$。

复杂度处理

在上述聚类算法中，通过一系列策略对时间复杂度和空间复杂度进行了优化。这些优化措施显著提高了算法处理大数据集的能力。下面详细描述这些优化及其对复杂度的影响。

时间复杂度的优化

1. **使用KD树优化近邻查询**：在大规模数据的近邻查询任务中，如果我们采取线性扫描的方式，需要计算查询点与每一个其他点的距离，这将导致时间复杂度为$O(N)$，这在数据量大的情况下是不可接受的。KD树（`K-Dimensional Tree`）是一种用于在k维度空间中存储和组织数据的高效数据结构，在很多情况下，KD树的查找效率远超于线性算法。KD树的生成是将数据空间逐次沿坐标轴划分，使每个数据点都被嵌入到一个特定的子空间。在代码中，KD树就被用作查找查询点的近邻，具体是用在`region_query`函数中。实际操作由`query_radius`方法完成，它找出KD树中与查询点距离在eps以内的所有点的索引，复杂度为$O(logN)$。
2. **避免重复查询**：维护了一个`visited`列表，记录哪些点已经被访问过。这样在查询近邻时，已访问过的点就可以跳过，避免了不必要的重复查询，减少了计算的次数。

通过上述优化，算法在处理数据集时的效率和可行性得到了显著提升。特别是使用KD树的策略，它不仅减少了查找查询点近邻的时间，也降低了存储距离信息所需的空间，从而优化了整个聚类算法的时间复杂度和空间复杂度。

可能的复杂度改进措施

时间复杂度优化

1. **并行化计算**：DBSCAN算法存在并行性，特别是在处理每一个簇时，可以使用一些工具例如`joblib`库将代码并行化进一步提升性能。
2. **内存优化**：如果数据集非常大，一次性将所有数据加载到内存可能会消耗过多资源。一种解决方法是使用迭代器一次只处理一个数据块，或者使用一些能有效处理大数据的工具，如`dask`库或`pandas`的`read_csv`函数中的`chunksize`参数。
3. **采样技术**：如果数据量仍然过大，使得时间或空间复杂性过高，那么可以考虑采样技术。你可以按一定策略（随机、分层、主动等等）选择数据的一个子集进行聚类。然后，根据这个子集的聚类结果，对剩余的数据进行标记。

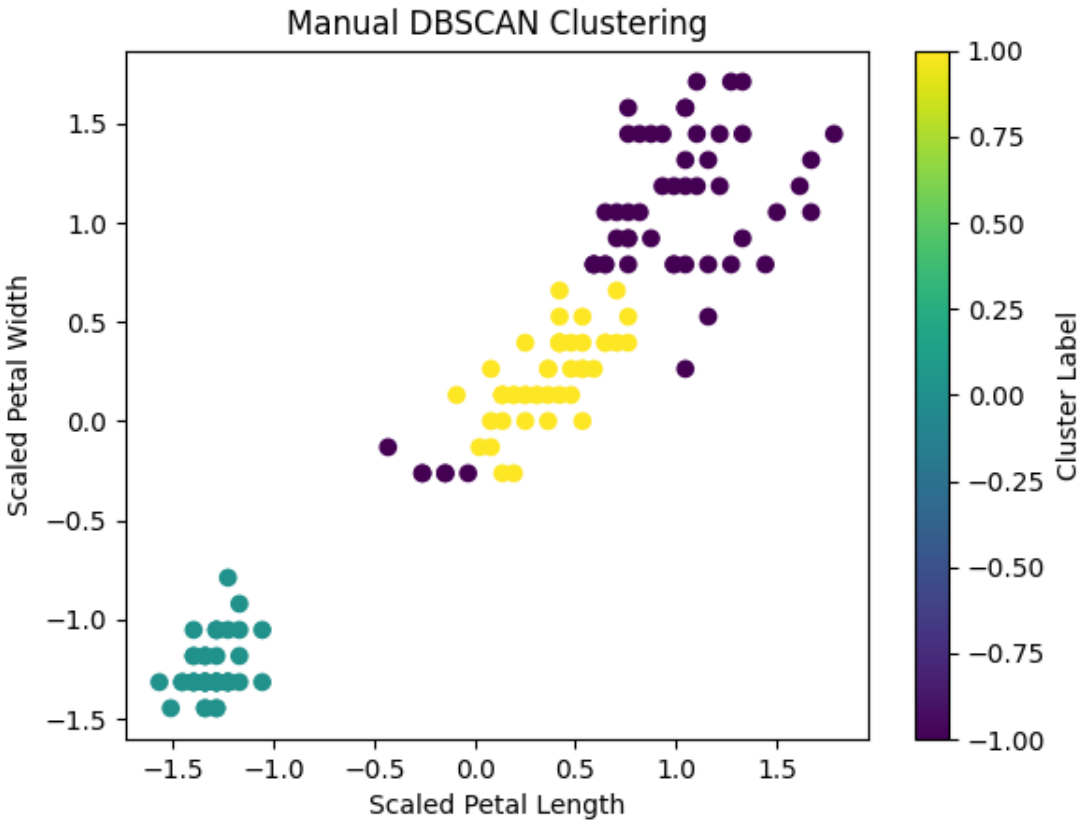
4. **向量化操作**：虽然你的代码中已经使用了KD树进行查询操作，但在`expand_cluster`函数中，仍然有一些Python的for循环和while循环。你可以考虑将这部分操作进一步向量化，比如`numpy`中的广播和其他向量化操作，来提高代码执行效率。

空间复杂度优化

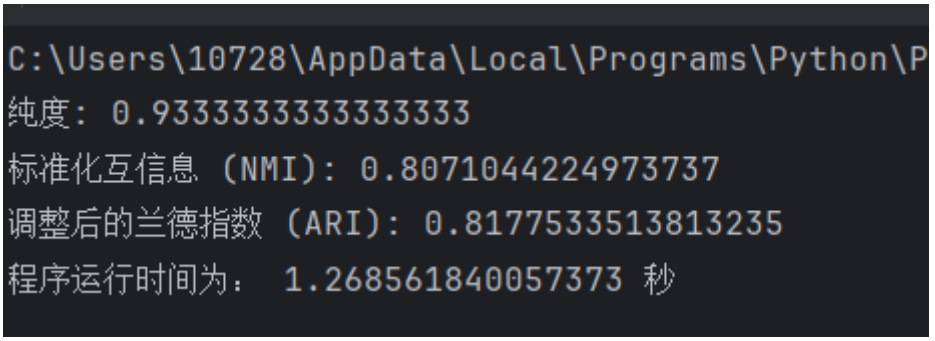
1. **数据类型调整**：根据数据的实际需求，选择适当的数据类型可以帮助减少空间占用。例如，如果你的数据集的特性允许（例如，数值范围较小或精度要求不高），你可以尝试将数据类型从默认的64位类型（如`float64`或`int64`）降低到32位或更低。
2. **按需加载数据**：如果你的数据集过大，不必要地将所有数据一次性加载到内存中可能会消耗大量资源。一种策略是使用生成器或迭代器逐块加载和处理数据。这种方式允许你只在需要时加载数据，从而减少内存开销。
3. **原位操作**：原位操作是一种减少内存开销的有效方式。一些函数或方法可以设置一个`inplace`函数参数来实现内存节省。在你的代码中，你可能使用到这些函数或方法的一些变种，它们会生成新的array返回，如`numpy.append`，你可以尝试寻找是否有其他原地操作的替代方法。
4. **高效的数据结构**：评估和优化数据结构也可以有效降低空间开销。例如，如果你有大量重复的数据，使用集（`set`）或类别（`category`）数据类型可能更有效率。或者，在保存稀疏数据时，使用稀疏矩阵而不是通常的`numpy`数组会更节省空间。

结果展示

散点图



评估指标



- 根据上述评估指标我们可以看出聚类所得结果较好，纯度较高