# 凝聚层次聚类（Agglomerative Clustering）

## 一、加载和预处理数据

首先，我选择了 Iris 数据集做聚类，可以从 UCI Machine Learning Repository 下载，我这里直接用 scikit-learn 来加载数据集。

```
from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_score
import time
import numpy as np
import pandas as pd

iris = load_iris() # 加载Iris数据集
X = iris.data  # 特征数据
```

Iris 数据集包含了 4 个属性：Sepal.Length（花萼长度）、Sepal.Width（花萼宽度）、Petal.Length（花瓣长度）、Petal.Width（花瓣宽度）。同时有 3 个种类：Iris Setosa（山鸢尾）、Iris Versicolour（杂色鸢尾），以及 Iris Virginica（维吉尼亚鸢尾）。

## 二、先使用 sklearn 库中自带的 AgglomerativeClustering 实现算法

AgglomerativeClustering 的构造函数中比较重要的参数有：

1、 n_clusters 参数，代表最终要分为多少个簇。

2、affinity 参数，起到定义距离的作用，有以下选项：
euclidean：欧式距离，最常用的距离度量。
manhattan：曼哈顿距离，计算的是在标准坐标系上的点之间的距离总和。
cosine：余弦相似度，常用于文本或高维数据的相似性度量。
precomputed：允许用户提供一个预先计算好的距离矩阵。

3、linkage 参数，用于计算两个簇之间的距离，共有 4 个选项:
ward：最小化合并聚类时的方差增加量。
average：合并两个簇所有成员间平均距离最小的两个簇。
complete:计算每一对簇中距离最远的两个样本的距离，并合并距离最远的两个样本所属簇。
single：计算每一对簇中距离最近的两个样本的距离，并合并距离最近的两个样本所属簇。

**实验设定：**

我将在 iris 数据集上进行 AgglomerativeClustering，并且尝试不同的 n_clusters，affinity 和 linkage 参数组合，以此来比较不同的组合对于运行效率和结果的影响。

```python
# 定义不同的参数组合
param_combinations = [
    (n, affinity, linkage)
    for n in [3, 5, 7, 10]  # 不同的n_clusters值
    for affinity in ['euclidean', 'manhattan', 'cosine'] # 不同的affinity度量
    for linkage in ['ward', 'complete', 'average', 'single'] # 不同的linkage策略
    if not (affinity != 'euclidean' and linkage == 'ward')  # 排除不兼容的组合
]

results = []
```

由于要事先定义，所以不使用 precomputed 距离；由于 manhattan 和 cosine 不能与 ward 链接策略一起使用，故略去；其他诸如切比雪夫距离和马氏距离因为不在选项内而忽略；对于结果评估，使用轮廓系数评估聚类效果的好坏，值越接近 1 表示聚类效果越好。

轮廓系数的计算方法如下：

- 对于每个样本，计算与同簇其他样本的平均距离 a。
- 对于每个样本，计算与最近簇内样本所在簇的平均距离 b。
- 计算轮廓系数：s= (b- a) / max(a, b)

对于每一个组合，打印出它的 runtime 和轮廓系数。

```python
# 对于每组参数，运行 AgglomerativeClustering 并记录结果
for n_clusters, affinity, linkage in param_combinations:

    start_time = time.time()
    clusterer = AgglomerativeClustering(n_clusters=n_clusters, affinity=affinity, linkage=linkage)

    labels = clusterer.fit_predict(X)
    end_time = time.time()

    # 计算轮廓系数
    score = silhouette_score(X, labels)

    # 记录结果
    results.append({
        'n_clusters': n_clusters,
        'affinity': affinity,
        'linkage': linkage,
        'runtime': end_time - start_time,
        'silhouette_score': score
    })

# 打印结果
for result in results:
    print(f"n_clusters: {result['n_clusters']}, Affinity: {result['affinity']}, Linkage: {result['linkage']}, Runtime: {result['runtime']:.4f}
```

```
n_clusters: 3, Affinity: euclidean, Linkage: ward, Runtime: 0.0010s, Silhouette Score: 0.5543
n_clusters: 3, Affinity: euclidean, Linkage: complete, Runtime: 0.0020s, Silhouette Score: 0.5136
n_clusters: 3, Affinity: euclidean, Linkage: average, Runtime: 0.0010s, Silhouette Score: 0.5542
n_clusters: 3, Affinity: euclidean, Linkage: single, Runtime: 0.0006s, Silhouette Score: 0.5121
n_clusters: 3, Affinity: manhattan, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.5544
n_clusters: 3, Affinity: manhattan, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.5535
n_clusters: 3, Affinity: manhattan, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.3403
n_clusters: 3, Affinity: cosine, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.3996
n_clusters: 3, Affinity: cosine, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.5538
n_clusters: 3, Affinity: cosine, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.5538
n_clusters: 5, Affinity: euclidean, Linkage: ward, Runtime: 0.0096s, Silhouette Score: 0.4844
n_clusters: 5, Affinity: euclidean, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.3462
n_clusters: 5, Affinity: euclidean, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.4307
n_clusters: 5, Affinity: euclidean, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.2838
n_clusters: 5, Affinity: manhattan, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.4734
n_clusters: 5, Affinity: manhattan, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.4318
n_clusters: 5, Affinity: manhattan, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.2194
n_clusters: 5, Affinity: cosine, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.1929
n_clusters: 5, Affinity: cosine, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.2656
n_clusters: 5, Affinity: cosine, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.3468
```

```
n_clusters: 7, Affinity: euclidean, Linkage: ward, Runtime: 0.0009s, Silhouette Score: 0.3422
n_clusters: 7, Affinity: euclidean, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.3298
n_clusters: 7, Affinity: euclidean, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.3707
n_clusters: 7, Affinity: euclidean, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.1328
n_clusters: 7, Affinity: manhattan, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.3066
n_clusters: 7, Affinity: manhattan, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.3570
n_clusters: 7, Affinity: manhattan, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.1474
n_clusters: 7, Affinity: cosine, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.0613
n_clusters: 7, Affinity: cosine, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.1416
n_clusters: 7, Affinity: cosine, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.1709
n_clusters: 10, Affinity: euclidean, Linkage: ward, Runtime: 0.0000s, Silhouette Score: 0.2925
n_clusters: 10, Affinity: euclidean, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.3029
n_clusters: 10, Affinity: euclidean, Linkage: average, Runtime: 0.0101s, Silhouette Score: 0.3083
n_clusters: 10, Affinity: euclidean, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.0251
n_clusters: 10, Affinity: manhattan, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.2792
n_clusters: 10, Affinity: manhattan, Linkage: average, Runtime: 0.0000s, Silhouette Score: 0.3305
n_clusters: 10, Affinity: manhattan, Linkage: single, Runtime: 0.0000s, Silhouette Score: 0.0688
n_clusters: 10, Affinity: cosine, Linkage: complete, Runtime: 0.0000s, Silhouette Score: 0.0436
n_clusters: 10, Affinity: cosine, Linkage: average, Runtime: 0.0091s, Silhouette Score: 0.0232
n_clusters: 10, Affinity: cosine, Linkage: single, Runtime: 0.0000s, Silhouette Score: -0.1080
```

**通过结果可以观察到：**

1、轮廓系数随着聚类簇数量的增加而整体下降。

2、euclidean 和 manhattan 距离在大多数情况下，提供了较高的轮廓系数；cosine 距离度量在聚类数增加时，其轮廓系数显著下降，特别是当聚类数为 10 时，部分情况下轮廓系数甚至为负值。

3、single 策略在大多数情况下提供了较低的轮廓系数；使用 euclidean 距离和 ward 策略似乎是一个较好的选择；complete 和 average 策略的表现较为一致，没有明显的优势，但 average 的时间复杂度可能会低一点。

由于 runtime 会受到多种因素的影响（如 CPU 负载、系统资源分配等），每次实际运行的结果可能会有所不同。时间复杂度可能可以提供一个更稳定的理解框架。

基础的凝聚层次聚类的时间复杂度为 $O(n^3)$，如果某个簇到其他所有簇的距离存放在一个有序表或堆中,层次聚类所需要的时间复杂度将为 $O(n^2 \log n)$

对于不同的 linkage 策略，时间复杂度大致如下：

- Ward, Average, Complete linkage: 这些策略的时间复杂度通常为 $O(n^2 \log n)$到 $O(n^3)$，具体取决于实现的细节。Ward 和 Average 链接在实践中通常表现为接近 $O(n^2 \log n)$，而 Complete 链接在某些情况下可能会接近 $O(n^3)$。
- Single linkage: 对于 single 链接策略，有一些特别高效的算法实现（如基于 MST 的算法），其时间复杂度可以达到 $O(n^2)$。

而不同的 affinity 对算法的总体时间复杂度影响较小，主要是影响单次距离计算的成本。

## 三、用代码实现凝聚层次聚类算法

我将使用在 iris 模型上表现较好的 euclidean 距离和 average 策略来实现该算法。

由于 sklearn 中的算法经过了合适的优化和聚类合并策略，运用连通性约束矩阵，减少了重复计算，所以本算法效果可能不如官方库好。

步骤：
1、初始距离计算、堆初始化：首先计算所有样本点之间的距离，并使用这些距离初始化一个最小堆。最小堆中的每个元素包括一个距离值和这个距离对应的两个样本点（或聚类）的索引。
2、聚类合并、距离更新：在每一步聚类过程中，算法从最小堆中弹出最小的距离及其对应的样本点（或聚类）对，作为要合并的聚类对。然后，根据这两个聚类的合并结果更新聚类索引，并重新计算新聚类与其他所有聚类之间的平均距离，将这些新的距离信息加入到最小堆中。
3、迭代：重复上述合并与距离更新步骤，直到聚类的数量减少到指定的目标聚类数目。

```
import numpy as np
from scipy.spatial.distance import pdist, squareform
import heapq

# 初始化链接矩阵和距离矩阵
def linkage_matrix_init(X):

    # 计算所有样本点之间的欧几里得距离
    dist_matrix = squareform(pdist(X, 'euclidean'))

    # 初始化链接矩阵，用于记录每一步聚类合并的信息
    linkage_matrix = np.zeros((len(X) - 1, 4))
    return dist_matrix, linkage_matrix

# 使用最小堆找到距离最近的两个聚类
def find_clusters_to_merge(heap, cluster_indices):

    while heap:
        dist, i, j = heapq.heappop(heap)
        if i in cluster_indices and j in cluster_indices and i != j:
            return (i, j), dist
    return (None, None), np.inf
```

```
def agglomerative_clustering(X, n_clusters):
    n_samples = X.shape[0]
    dist_matrix, linkage_matrix = linkage_matrix_init(X)
    cluster_indices = {i: {i} for i in range(n_samples)}

    # 初始化最小堆
    heap = []
    for i in range(n_samples):
        for j in range(i + 1, n_samples):
            heapq.heappush(heap, (dist_matrix[i, j], i, j))

    for step in range(n_samples - n_clusters):
        # 找到要合并的两个聚类
        (i, j), dist = find_clusters_to_merge(heap, cluster_indices)

        # 合并聚类
        new_cluster = cluster_indices[i].union(cluster_indices[j])
        new_index = n_samples + step

        # 更新链接矩阵
        linkage_matrix[step, 0] = i
        linkage_matrix[step, 1] = j
        linkage_matrix[step, 2] = dist
        linkage_matrix[step, 3] = len(new_cluster)

        # 更新聚类索引
        del cluster_indices[i], cluster_indices[j]
        cluster_indices[new_index] = new_cluster
```

```
        # 更新堆中的距离
        for k in cluster_indices.keys():
            if k != new_index:
                new_dist = np.mean(dist_matrix[np.ix_(list(new_cluster), list(cluster_indices[k]))])
                heapq.heappush(heap, (new_dist, new_index, k))

    # 生成最终的聚类标签
    cluster_labels = np.zeros(n_samples, dtype=int)
    for label, indices in cluster_indices.items():
        for index in indices:
            cluster_labels[index] = label

    return cluster_labels, linkage_matrix

# 示例
from sklearn.datasets import load_iris
from sklearn.metrics import silhouette_score

iris = load_iris()
X = iris.data

# 执行聚类
labels, matrix = agglomerative_clustering(X, 5)

# 评估聚类结果
score = silhouette_score(X, labels)
print(f"Silhouette Score: {score:.4f}")
```

算法思想：通过引入最小堆来维护距离，每次从堆中取出最小的距离对应的聚类对进行合并，从而减少每次查找最近聚类对的时间复杂度。当合并两个聚类后，需要更新这两个聚

类与其他所有聚类之间的平均距离，并将这些距离加入到最小堆中，以保证下一次能够正确地找到最近的聚类对。

时间复杂度优化：

使用最小堆来管理聚类间的距离可以显著提高查找最近聚类对的效率。在未优化的情况下，每次查找最近聚类对的时间复杂度为 $O(n^2)$，其中 n 为当前聚类的数量。而最小堆使得该操作的时间复杂度降低到了 O(log n)，因为堆的插入和弹出操作的时间复杂度为 O(log n)。

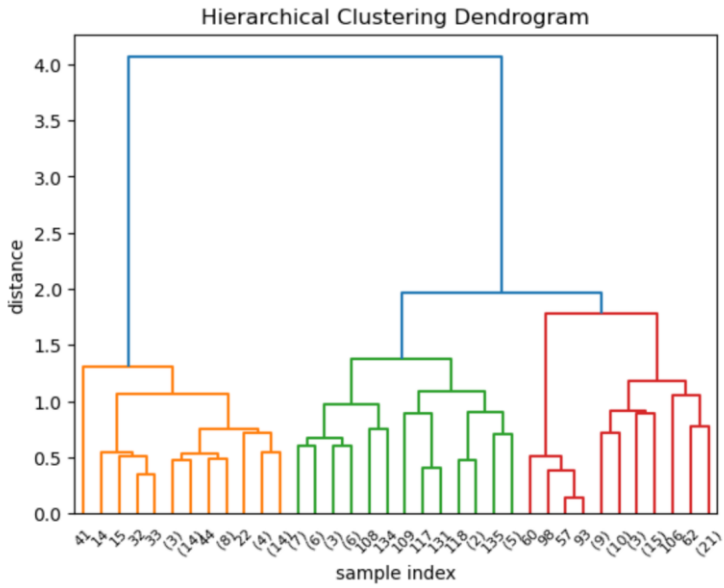
空间复杂度优化：

主要体现在对距离矩阵的处理上。在初始的距离计算之后，算法不需要存储完整的距离矩阵，因为聚类间的距离被动态计算并即时更新在最小堆中。虽然最小堆需要额外的空间来存储距离和索引，但这比维护完整的距离矩阵要小得多，特别是随着聚类合并的进行，聚类数量减少，相关的距离条目也相应减少。

结果与使用官方库得到的结果相同（n_clusters=5，affinity=euclidean，linkage=average）：

Silhouette Score: 0.4307

树状图：




可以看出有 3 个簇的区别。

## 四、凝聚层次聚类的优缺点

1、优点

1）一次性地得到了整个聚类的过程，改变 cluster 数目不需要再次计算数据点的归属。

2）适用于任意形状的聚类，并且对样本的输入顺序不敏感。

2、缺点

1）时间复杂度大。

2）由于运用贪心算法，得到的是局部最优，不一定就是全局最优。