聚类算法

(K-Means)

实验报告

计算机科学与技术学院

2352018 刘彦

2024 年 12 月 9 日

一、 实验目的和内容

1. 理解 K-Means 聚类算法的基本原理

掌握 K-Means 算法中包括初始化、分配簇标签、更新质心、迭代优化等关键步骤，了解其收敛条件和适用场景，分析不同参数（如簇的数量、标准差、初始质心等）对聚类效果的影响。在不依赖库函数的情况下，通过编程实现 K-Means 聚类算法，加深对其算法流程的理解。

2. 生成和处理模拟数据集

学习如何通过程序生成具有特定分布的数据集，并将其保存为外部文件（如 CSV 格式）以便加载和处理。

3. 结果可视化验证聚类效果

通过 K-Means 聚类算法对生成的数据集进行聚类，分析算法在数据点分布明确的数据集上的效果。观察每次迭代中簇标签和质心的变化，直观理解 K-Means 的优化过程。

二、 实验过程

1. 实验环境准备

确保已安装 python 环境，并配置必要的依赖库（如 numpy、matplotlib 等）。利用自行编写的 set.py（附在文后）准备好实验所需的 csv 数据文件，通过设置参数得到不同类型的测试数据集。

```
# 数据集的参数
n_samples = 300  # 数据点总数
n_features = 2  # 特征数（二维数据）
n_clusters = 4  # 聚类簇的数量
random_state = 15  # 随机种子，用于结果可复现
cluster_std = 1.9  # 每个簇的标准差（控制数据点分布的紧密程度）
```

2. 主要算法介绍

① 算法基本步骤

- 初始化：随机选择 K 个点作为初始质心（簇中心）。质心的初始选择可以影响算法的收敛速度和聚类质量。
- 分配簇标签：对每个数据点，计算其到所有质心的欧几里得距离。将数据

点分配到距离最近的质心所属的簇。

- 更新质心：根据当前簇的分配，重新计算每个簇的质心（均值）

$$\mu_k = \frac{1}{|C_k|}\sum_{x \in C_k} x_i$$

其中，μ_k是簇C_k的新质心，$|C_k|$是簇中数据点的数量。

- 迭代优化：重复“分配簇标签”和“更新质心”两个步骤，直到满足以下任一条件：质心的位置不再发生变化（收敛）。达到最大迭代次数。

② 算法优化目标

该算法的目标是最小化如下目标函数：

$$J = \sum_{k=1}^{K}\sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

③ 算法复杂度

- 时间复杂度：$O(n \cdot K \cdot d \cdot t)$
- 空间复杂度：$O(n + K + d)$

其中，n 为数据点数，K 为簇的数量，d 为每个数据点的特征数，t 为迭代次数。

3. 主要函数说明

在 class KMeansCustom 中，主要函数如下：

① __init__函数

初始化 K-Means 算法的基本参数。

```
def __init__(self, n_clusters, max_iter=300, tol=1e-4):
    """
    初始化 K-Means 模型
    :param n_clusters: 聚类簇数
    :param max_iter: 最大迭代次数
    :param tol: 收敛容忍度
    """
    self.n_clusters = n_clusters
    self.max_iter = max_iter
```

② fit 函数

执行 K-Means 聚类过程，训练模型。

```
def fit(self, X):
    """
    训练模型
    :param X: 输入数据集（二维数组）
    """
    n_samples, n_features = X.shape
    # 随机初始化质心
    random_idx = np.random.permutation(n_samples)[:self.n_clusters]
    self.centroids = X[random_idx]
    for i in range(self.max_iter):
        # 分配每个点到最近的质心
        self.labels = self._assign_clusters(X)
        # 可视化当前迭代的聚类结果
        self._plot_iteration(X, i)
        # 更新质心
        new_centroids = self._update_centroids(X)
        # 检查是否收敛
        if np.linalg.norm(self.centroids - new_centroids, axis=None) < self.tol:
            break
        self.centroids = new_centroids
```

③ _assign_clusters 函数

为每个数据点分配簇标签，找到每个点到各质心的最小距离。返回每个数据点所属的簇标签（索引）。

```
def _assign_clusters(self, X):
    """
    计算每个点到质心的距离并分配到最近的质心
    :param X: 输入数据集
    :return: 每个点对应的簇标签
    """
    distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
    return np.argmin(distances, axis=1)
```

④ _update_centroids 函数

根据每个簇中的数据点，计算新的质心。

```
def _update_centroids(self, X):
    """
    更新每个簇的质心为簇内所有点的均值
    :param X: 输入数据集
    :return: 更新后的质心
    """
    centroids = np.zeros((self.n_clusters, X.shape[1]))
    for k in range(self.n_clusters):
        points = X[self.labels == k]  # 获取属于簇 k 的所有点
        if len(points) > 0:
            centroids[k] = points.mean(axis=0)  # 计算均值
    return centroids
```

⑤ _plot_iteration 函数

调 matplotlib 库，可视化每次迭代的结果，包括当前数据点的簇分配和质心位置。

```
def _plot_iteration(self, X, iteration):
    """
    绘制每次迭代的聚类过程
    :param X: 输入数据集
    :param iteration: 当前迭代次数
    """
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=self.labels, s=50,
cmap='viridis', label='Data Points')
    plt.scatter(self.centroids[:, 0], self.centroids[:, 1],
c='red', s=200, alpha=0.75, label='Centroids')
    plt.title(f"Iteration {iteration + 1}")
    plt.xlabel("Feature x")
    plt.ylabel("Feature y")
    plt.legend()
    plt.grid(True)
    plt.show()
```

⑥ predict 函数

为新的数据点分配簇标签。

```
def predict(self, X):
    """
    根据模型预测数据的簇标签
    :param X: 输入数据集
    :return: 每个点对应的簇标签
    """
    return self._assign_clusters(X)
```

三、 测试样例说明

1. 数据来源

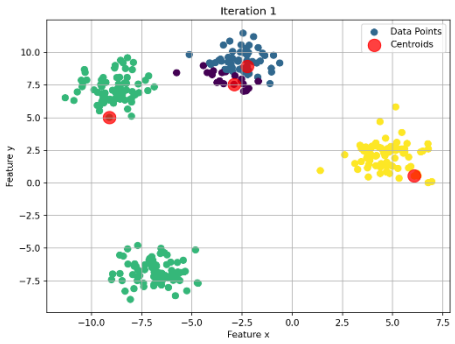
测试样例的数据集来源于利用自行编写的 set.py（附在文后）准备好实验所需的 csv 数据文件，其中存储了散点的坐标。

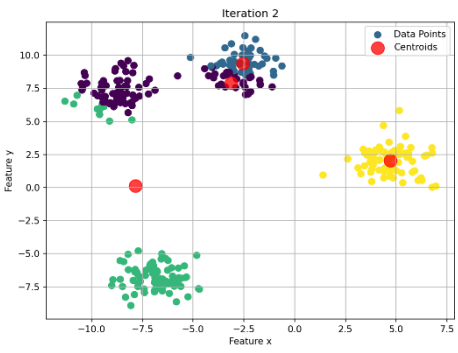
2. 数据准备

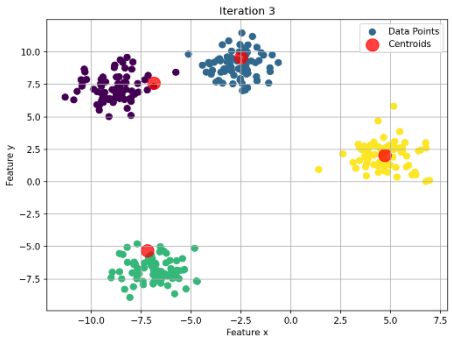
该数据集共有 3 个 csv 文件，散点的分散程度不同，测试该算法在不同条件下的实现情况。

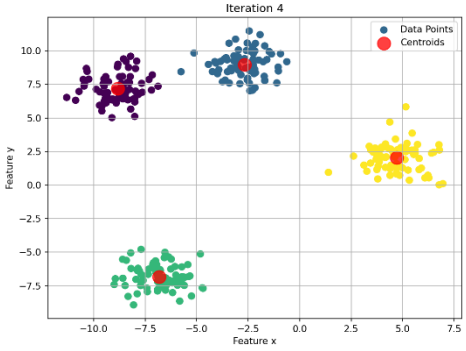
四、 测试结果及说明

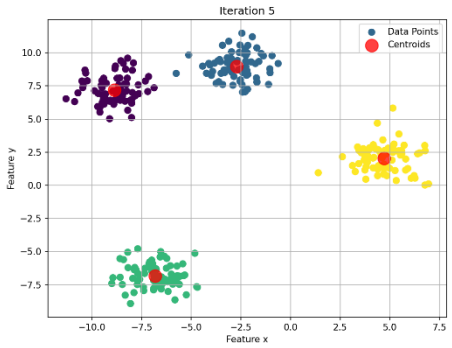
1. 测试结果











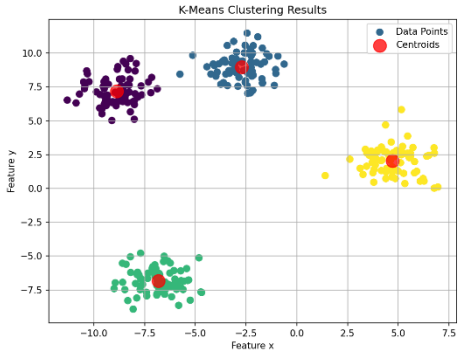
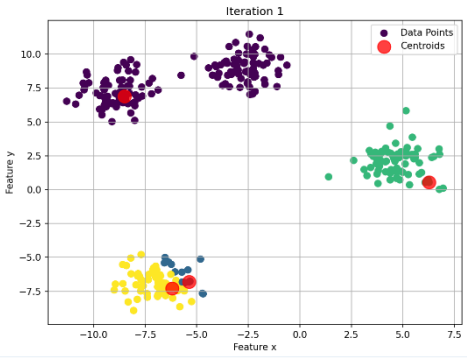
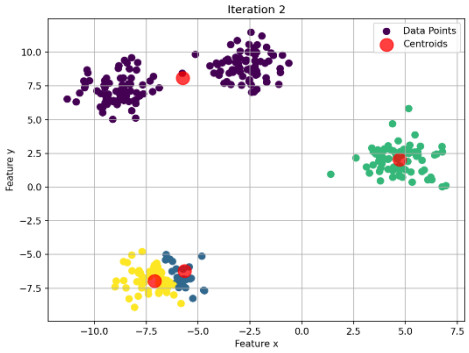
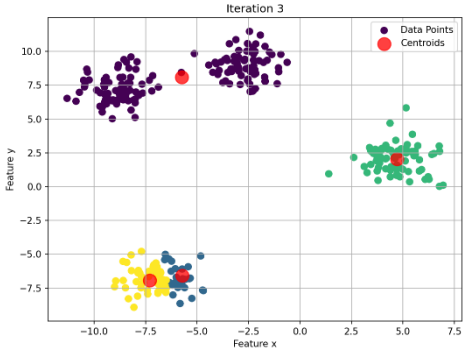
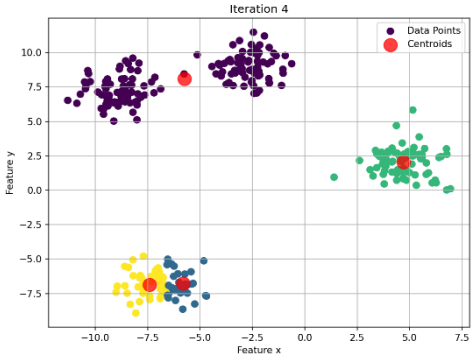


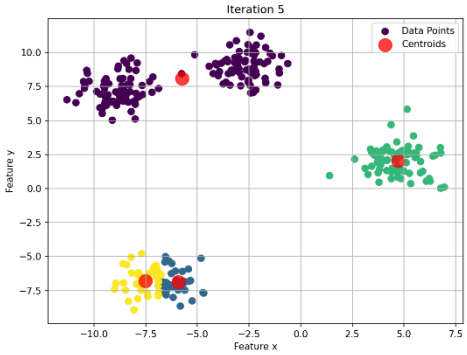
图 1 第 1 个数据集的实验结果一

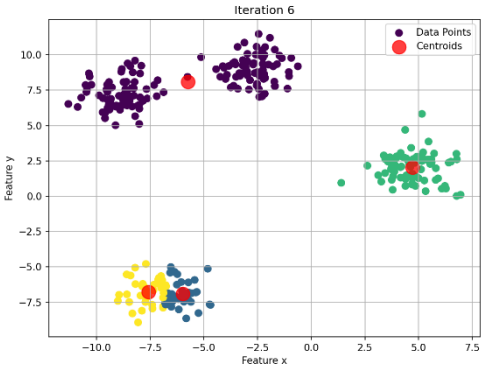












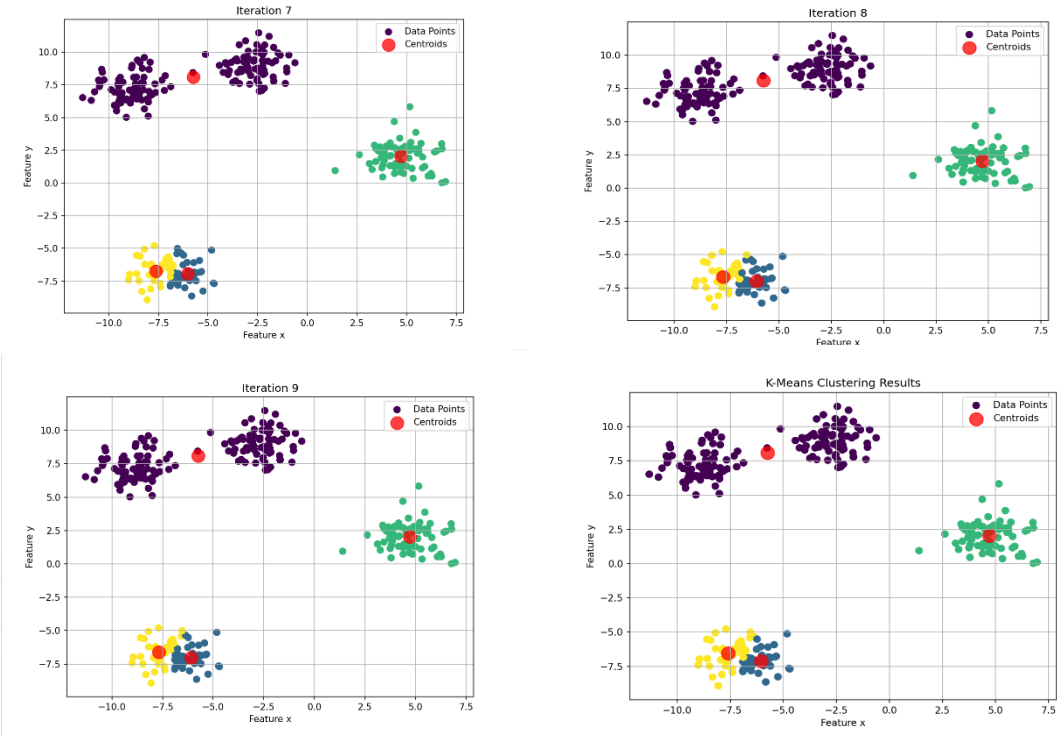


图 2 第 1 个数据集的实验结果二

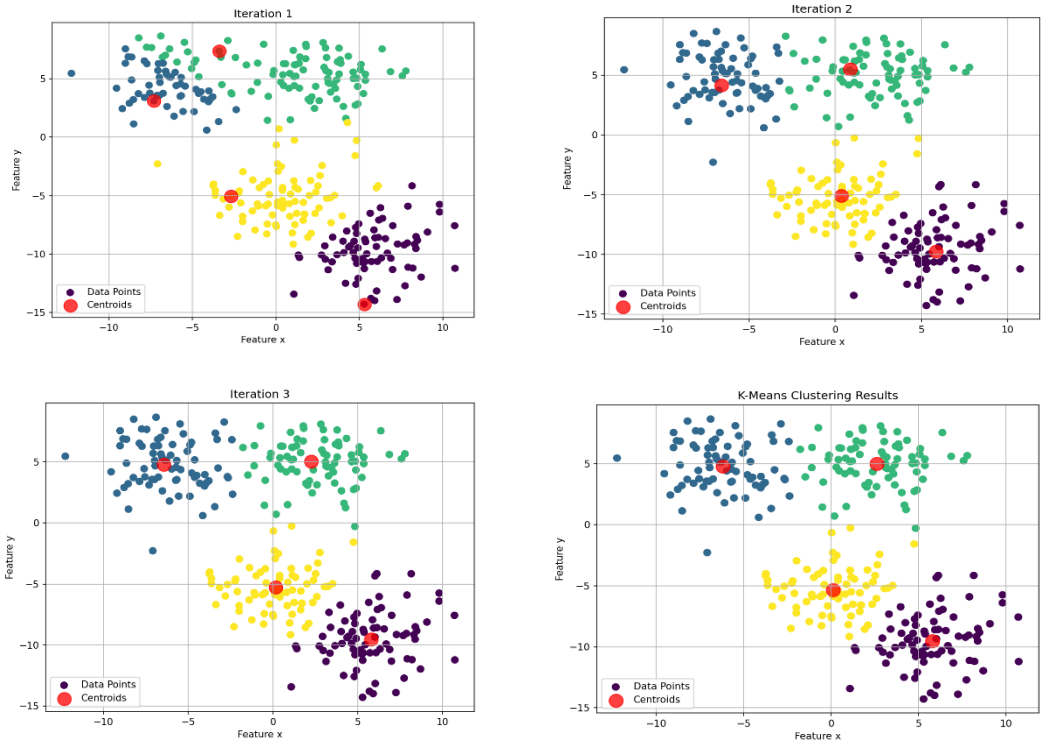
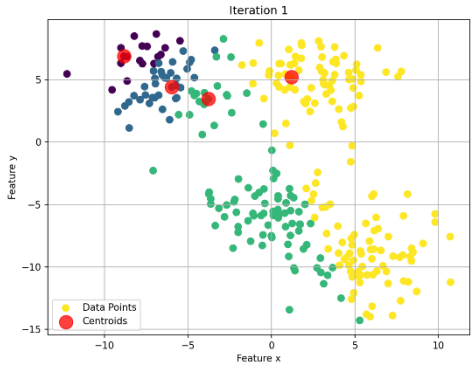
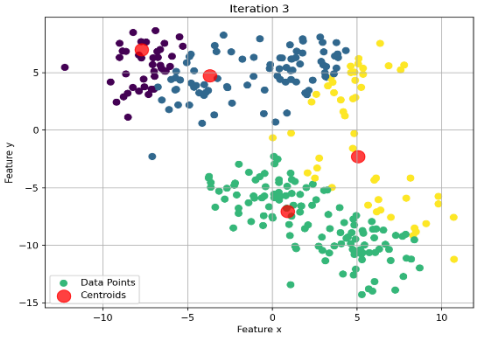
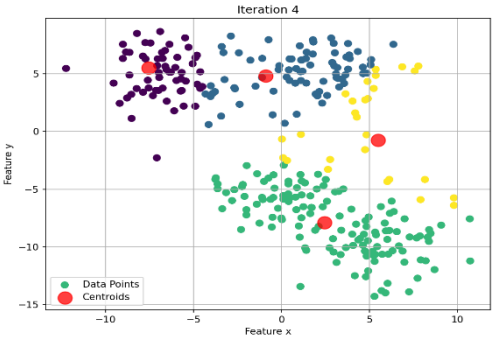


图 3 第 2 个数据集的实验结果一

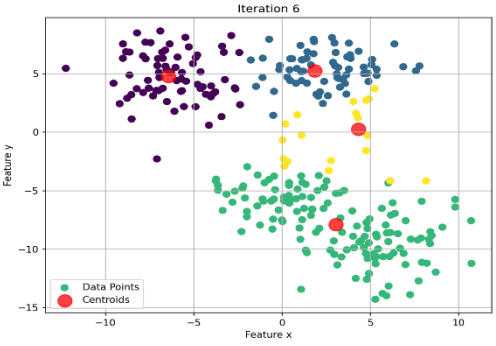

Iteration 1
Feature y
Feature x
Data Points
Centroids

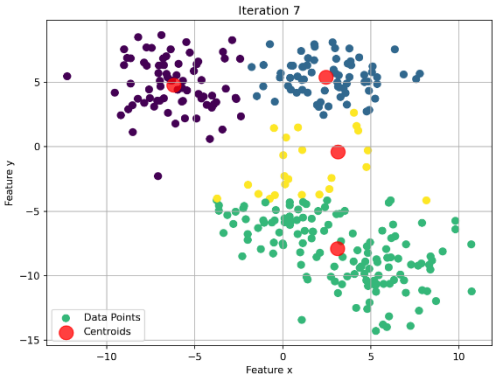

Iteration 2
Feature y
Feature x
Data Points
Centroids

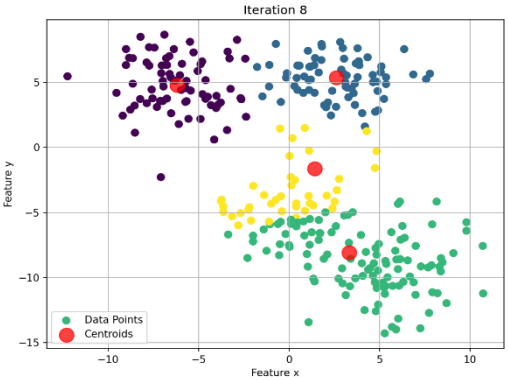

Iteration 3
Feature y
Feature x
Data Points
Centroids


Iteration 4
Feature y
Feature x
Data Points
Centroids


Iteration 5
Feature y
Feature x
Data Points
Centroids


Iteration 6
Feature y
Feature x
Data Points
Centroids


Iteration 7
Feature y
Feature x
Data Points
Centroids


Iteration 8
Feature y
Feature x
Data Points
Centroids

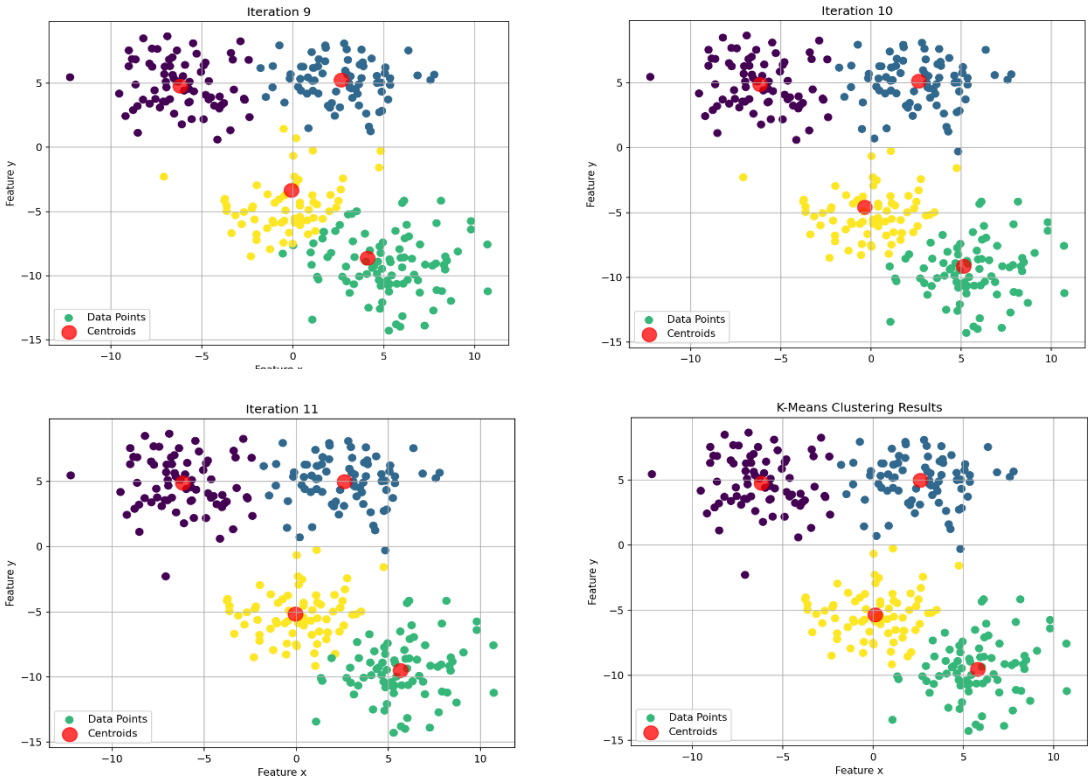
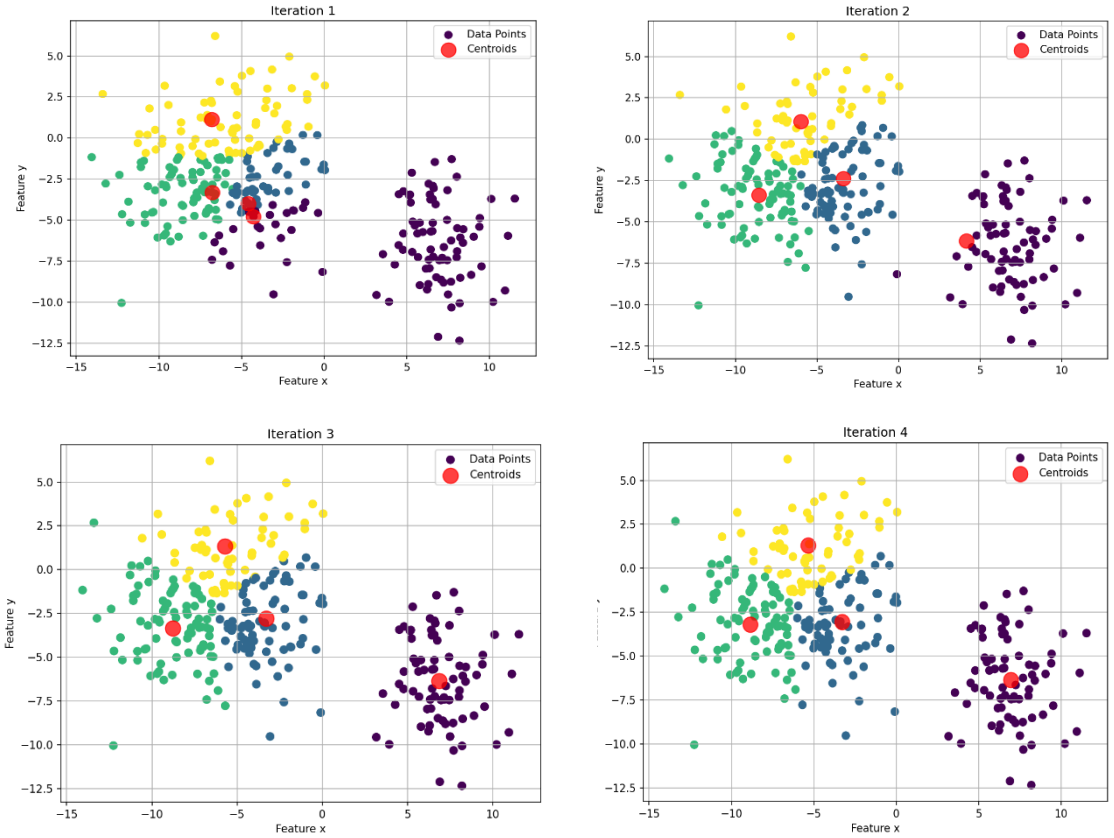


图 4 第 2 个数据集的实验结果二



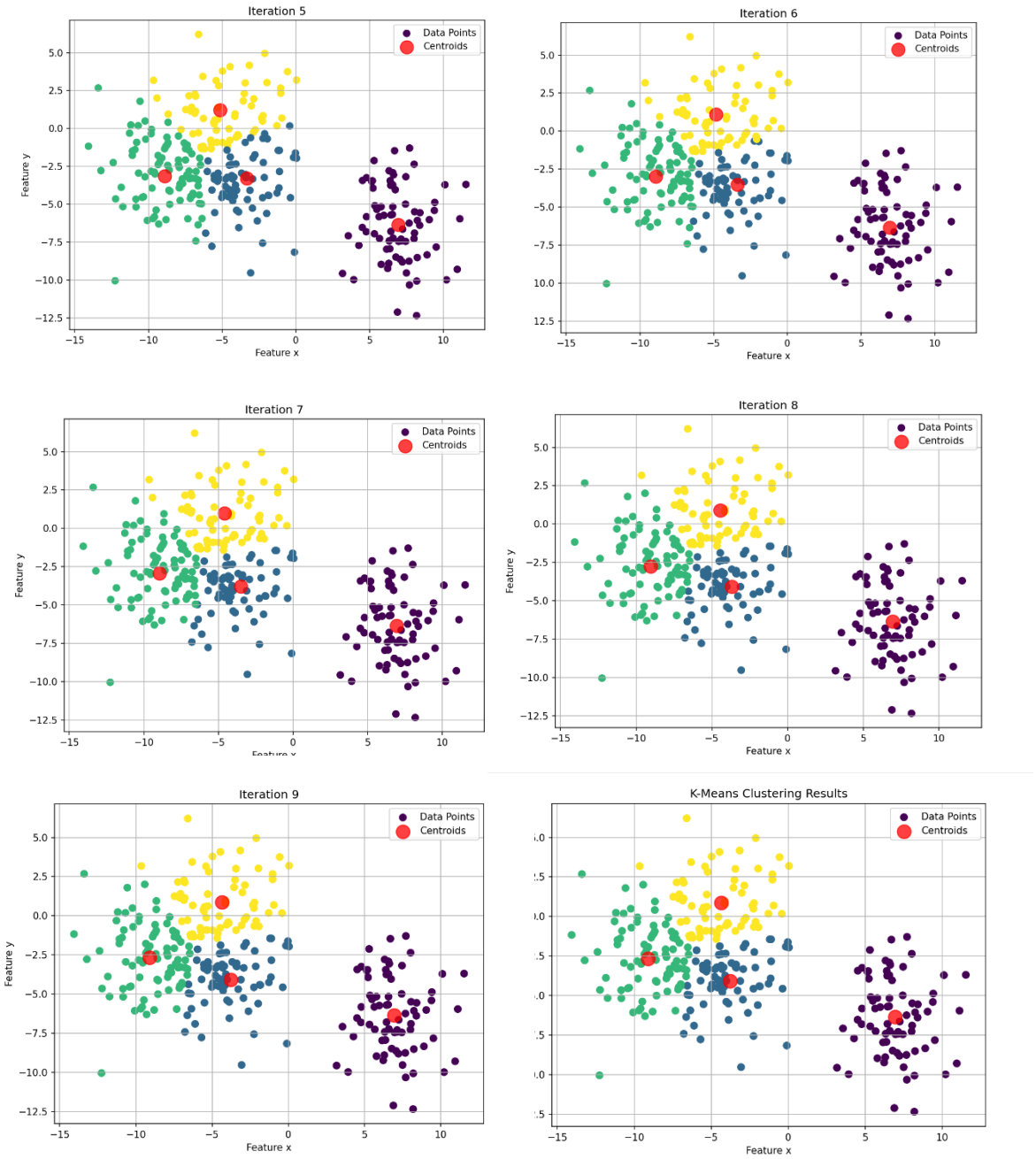


图 5 第 3 个数据集的实验结果

2. 结果说明

数据点被分配到距离其质心最近的簇，理想情况下，簇内的数据点应该是紧密集中的，簇之间的边界则应该相对清晰。就实验结果来看，该算法可以完成聚类任务，但仍存在找出局部最优解的情况（如图 2），说明此算法还存在一定问题。

另外，经过观察，散点分散程度越大，找出局部最优解的可能性降低，该算

法会逐步调整纠错（如图 5），但在簇的边界较为清晰时不容易从局部最优解中走出。

3. 参数的影响

①簇的数量（K）

在实际应用中，要选择合适的 K 值。如果选择的簇数过多，会导致过拟合，使得每个簇内部的数据点过于分散；如果簇数过少，则可能无法有效区分数据的不同特征。

②标准差

标准差影响簇的密度。较小的标准差会让数据点更加集中，从而使得簇之间的边界更加清晰，聚类结果可能更好。

④ 初始质心的选择

随机初始化质心可能会影响结果。某些情况下，质心的初始化位置不佳可能导致局部最优解，尤其是在数据分布较为集中时。

五、 改进之处

在上述实验后，我发现随机初始化质心可能会影响结果。某些情况下，质心的初始化位置不佳可能导致局部最优解。为此，我对代码做出了相应的调整，对质心的初始化方法做了修改。由原来随机选取，到现在：

- 选择数据点中的一个随机点作为第一个质心。
- 根据每个数据点到已选择质心的最小距离，计算该点的距离的平方，并按这个平方的比例选择下一个质心。即距离较远的点更有可能被选择为质心。
- 重复以上过程，直到选择出所有的质心。

经过再次实验，发现该算法能够较好地选择初始质心，从而使得聚类结果更加稳定。且初始质心位置通常较分散，通常会使迭代次数变少。

```python
def _kmeans_set_centroids(self, X):
    """
    初始化质心
    :param X: 输入数据集
    :return: 初始化的质心
    """
    n_samples = X.shape[0]
    centroids = []
    # Step 1: 随机选择第一个质心
    centroids.append(X[np.random.choice(n_samples)])
    # Step 2: 选择其余的质心
    for _ in range(1, self.n_clusters):
        distances = np.min(np.linalg.norm(X[:, np.newaxis] -
np.array(centroids), axis=2), axis=1)
        probabilities = distances ** 2
        probabilities /= probabilities.sum()  # 正常化为概率分布
        next_centroid = X[np.random.choice(n_samples,
p=probabilities)]
        centroids.append(next_centroid)
    return np.array(centroids)
```

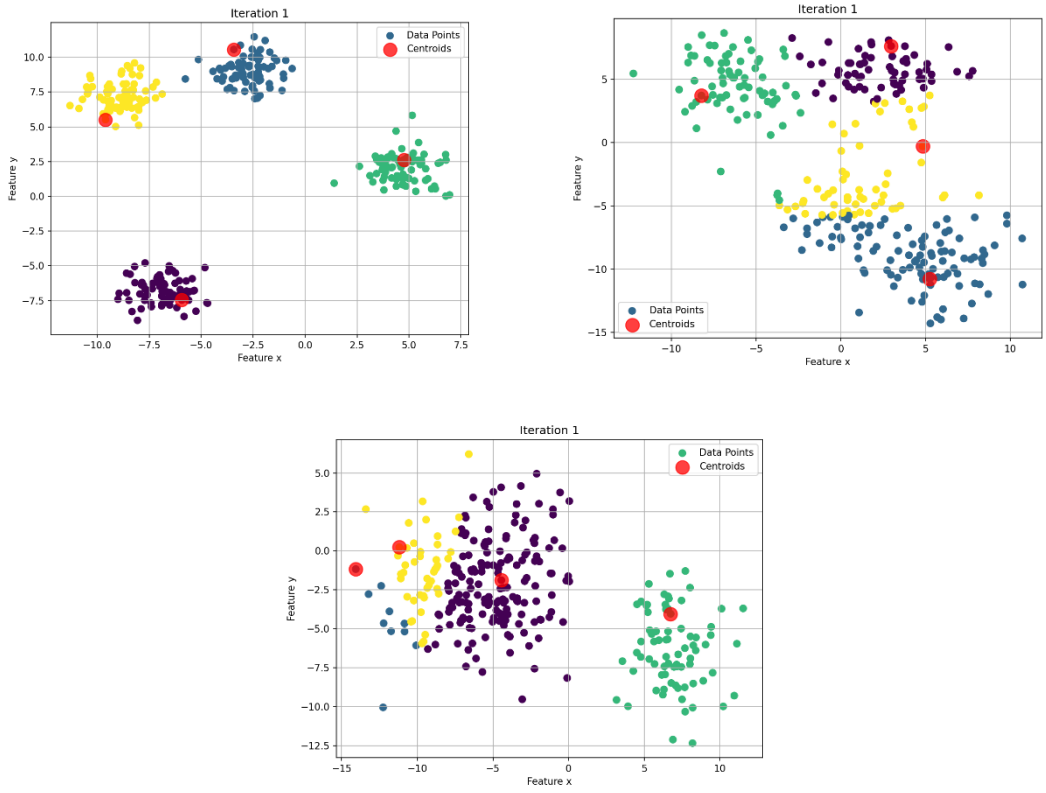


图 6 改进后三个数据集的初始质点位置

六、 心得体会

1. 对聚类算法有了更深入的理解

K-Means的核心思想是基于欧几里得距离的最小化，通过迭代优化目标函数，找到数据点的最优分组。这种方法简单而高效，是无监督学习中最经典的算法之一。

改进后解决了随机初始化质心可能导致局部最优的问题，通过更加分散的初始质心选择，提高了聚类效果和收敛速度。这让我理解到，初始化对于算法性能的影响是至关重要的。

2. 从理论到代码实现的转化

自己从零实现 K-Means 的代码，经历了从数学公式到编程逻辑的转化过程。这种实践让我明白了聚类算法的每一个步骤（质心初始化、簇分配、质心更新）的细节，以及它们在代码中的具体表现形式。

3. 数据可视化的重要性

实验中通过 Matplotlib 对每次迭代的聚类结果进行动态可视化，不仅帮助我理解了算法的内部工作原理，也让最终的实验结果更加直观。数据可视化在算法调试和结果分析中起到了至关重要的作用，有助于快速发现问题，比如初始质心选择是否合理等。

4. 改进和扩展的方向

可以尝试将 K-Means 与其他聚类算法（如密度聚类 DBSCAN、谱聚类等）进行对比，探索不同数据分布下的最优选择。

七、 附录：主要源代码

k-means.py

```
import numpy as np
import matplotlib.pyplot as plt
import csv

# 第一步：从 csv 文件中加载数据集
data_filename = "dataset.csv"  # 数据文件名
X = []
with open(data_filename, mode='r') as file:
```

```
    reader = csv.reader(file)
    next(reader)  # 跳过表头
    for row in reader:
        X.append([float(value) for value in row])  # 将每一行数据转换为浮点数并添加到列表中
X = np.array(X)  # 将列表转换为 NumPy 数组
print(f"读入数据库：{data_filename}中的数据")  # 打印提示信息
# 第二步：定义自定义的 K-Means 聚类算法
class KMeansCustom:
    略，具体实现放在【主要函数说明】中
# 第三步：应用自定义的 K-Means 聚类算法
n_clusters = 4  # 聚类簇数
kmeans = KMeansCustom(n_clusters=n_clusters)
kmeans.fit(X)  # 训练模型
y_kmeans = kmeans.predict(X)  # 获取聚类结果
# 第四步：最终聚类结果的可视化
plt.figure(figsize=(8, 6))
# 绘制数据点，用簇标签的颜色区分
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis', label='Data Points')
# 绘制质心
centers = kmeans.centroids
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, label='Centroids')
plt.title("K-Means Clustering Results")
plt.xlabel("Feature x")
plt.ylabel("Feature y")
plt.legend()
plt.grid(True)
plt.show()
```

k-means-v2.py

```
import numpy as np
import matplotlib.pyplot as plt
import csv

# 第一步：从 CSV 文件中加载数据集
data_filename = "dataset.csv"  # 数据文件名
X = []
with open(data_filename, mode='r') as file:
    reader = csv.reader(file)
    next(reader)  # 跳过表头
    for row in reader:
        X.append([float(value) for value in row])  # 将每一行数据转换为浮
```

```
点数并添加到列表中
X = np.array(X)  # 将列表转换为 NumPy 数组
print(f"读入数据库：{data_filename}中的数据")  # 打印提示信息
# 第二步：自定义 K-Means 聚类算法
class KMeansCustom:
    def _kmeans_set_centroids(self, X):
        """
        初始化质心
        :param X: 输入数据集
        :return: 初始化的质心
        """
        n_samples = X.shape[0]
        centroids = []
        # Step 1: 随机选择第一个质心
        centroids.append(X[np.random.choice(n_samples)])
        # Step 2: 选择其余的质心
        for _ in range(1, self.n_clusters):
            distances = np.min(np.linalg.norm(X[:, np.newaxis] -
np.array(centroids), axis=2), axis=1)
            probabilities = distances ** 2
            probabilities /= probabilities.sum()  # 正常化为概率分布
            next_centroid = X[np.random.choice(n_samples,
p=probabilities)]
            centroids.append(next_centroid)
        return np.array(centroids)
    略，具体实现放在【主要函数说明】中
# 第三步：应用 K-Means++ 聚类算法
n_clusters = 4  # 聚类簇数
kmeans_plus_plus = KMeansCustom(n_clusters=n_clusters)
kmeans_plus_plus.fit(X)  # 训练模型
y_kmeans_plus_plus = kmeans_plus_plus.predict(X)  # 获取聚类结果
# 第四步：最终聚类结果的可视化
plt.figure(figsize=(8, 6))
# 绘制数据点，用簇标签的颜色区分
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans_plus_plus, s=50,
cmap='viridis', label='Data Points')
# 绘制质心
centers = kmeans_plus_plus.centroids
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75,
label='Centroids')
plt.title("K-Means++ Clustering Results")
plt.xlabel("Feature x")
plt.ylabel("Feature y")
plt.legend()
```

```
plt.grid(True)
plt.show()
```

set.py

```
import numpy as np
import csv

# 数据集的参数
n_samples = 300  # 数据点总数
n_features = 2  # 特征数（二维数据）
n_clusters = 4  # 聚类簇的数量
random_state = 15  # 随机种子，用于结果可复现
cluster_std = 1.9  # 每个簇的标准差（控制数据点分布的紧密程度）
# 设置随机种子以确保每次生成的数据集一致
np.random.seed(random_state)
# 随机生成聚类簇的中心点
centers = np.random.uniform(-10, 10, (n_clusters, n_features))  # 在[-
10, 10]范围内生成簇中心
# 根据簇中心生成数据点，数据点围绕中心随机分布
X = []  # 初始化数据集列表
for center in centers:
    # 在每个中心点周围生成 n_samples // n_clusters 个数据点
    X.append(center + cluster_std * np.random.randn(n_samples //
n_clusters, n_features))
X = np.vstack(X)  # 将数据点堆叠为二维数组
# 将数据集保存到 CSV 文件中
data_filename = "dataset.csv"  # 文件名
with open(data_filename, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Feature1", "Feature2"])  # 写入表头
    writer.writerows(X)  # 写入数据点
print(f"包含 {n_samples} 个样本和 {n_features} 个特征的数据集已保存到
'{data_filename}'")  # 打印提示信息
```