同济大学

TONGJI UNIVERSITY

机器学习实验报告

报告名称	回归实验
组　　长	2152118　史君宝
组　　员	1952087　王鑫伟
	2151520　王禹皓
	2153812　彭兆祥
任课教师	李洁
日　　期	2023 年 12 月 16 日

目录

同济大学

一、 课题说明

小组成员分工

- 1952087 王鑫伟　负责不同原型聚类的结果分析
- 2151520 王禹皓　负责 kmeans 聚类的实现，整理资料撰写课题报告
- 2152118 史君宝　负责数据的预处理，课题汇报
- 2153812 彭兆祥　负责层次聚类的实现

二、 实验过程

1. 数据准备

数据集来源：Click Here! 这个数据集与不同葡萄酒的特性有关，实验的目标是根据这些葡萄酒的特性对葡萄酒的种类进行分类。

1. 数据结构：数据集包含 178 行，每行代表一个不同的葡萄酒样本。
2. 特征：
    - Alcohol：酒精含量。
    - Malic_Acid：苹果酸含量。
    - Ash：灰分含量。
    - Ash_Alcanity：灰分的碱性。
    - Magnesium：镁含量。
    - Total_Phenols：总酚含量。
    - Flavanoids：黄烷醇含量。
    - Nonflavanoid_Phenols：非黄烷醇酚含量。
    - Proanthocyanins：原花青素含量。
    - Color_Intensity：颜色强度。
    - Hue：葡萄酒的色调。
    - OD280：稀释葡萄酒的 OD280/OD315 值。
    - Proline：脯氨酸含量。
3. 统计摘要：
    - 每个特征的平均值不同，表明葡萄酒的化学成分具有多样性。例如，平均酒精含量约为 13%，而平均脯氨酸含量约为 747。
    - 标准差（std）值显示了数据集的可变性，这对于聚类分析至关重要。例如，镁含量的标准差约为 14.28，表明不同葡萄酒中镁水平的分布相当广泛。

这个数据集非常适合进行聚类分析，因为它包含可以用来根据其化学性质对相似葡萄酒进行分组的多个变量。聚类可能揭示基于成分在葡萄酒中的有趣模式或分组。

2. 数据预处理

(1) 将原来的数据转存到 csv 文件中原来的数据是保存在一个.data 文件中，我们需要将其转存到 csv 文件中。

```
import csv

# 打开数据文件
file = open('wine.data', 'r')
csvfile = open('data.csv', 'w', newline='')

# 创建CSV写入器
writer = csv.writer(csvfile)

# 写入表头
writer.writerow(['class', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash', 'Magnesium'
, 'Total phenols',
                 'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity', 'Hue',
                 'OD280/OD315 of diluted wines', 'Proline'])

# 逐行读取数据文件
for line in file:
    # 去除行末尾的换行符，并按逗号分隔数据
    data = line.strip().split(',')

    # 将数据写入CSV文件
    writer.writerow(data)

# 关闭文件
file.close()
csvfile.close()
```

(2) 查看数据的基本信息

```
import pandas as pd

train_df = pd.read_csv("data.csv")
print(train_df.info())
```

(3) 查看数据的统计信息

```
# 删除"class"列
train_df_without_class = train_df.drop("class", axis=1)

# 打印描述统计
print(train_df_without_class.describe())
```

(4) 绘制各个数据的条形图分布

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(3, 1, figsize=(10, 6))
```

```
    plt.subplots_adjust(top=2)

    ax[0].hist(train_df['Alcohol'], color='#10A37F', bins=50, edgecolor='black')
    ax[0].set_title('Alcohol')

    ax[1].hist(train_df['Malic acid'], color='#10A37F', bins=50, edgecolor='black')
    ax[1].set_title('Malic acid')

    ax[2].hist(train_df['Ash'], color='#10A37F', bins=50, edgecolor='black')
    ax[2].set_title('Ash')

    plt.show()
```

(5) 具体的处理数据

```
    # 查看数据的缺失率
    print(train_df.isnull().sum().sort_values(ascending = False) / train_df.shape[0])

    # 数据归一化
    from sklearn.preprocessing import StandardScaler

    cols = ['class', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash', 'Magnesium', 'Total phenols',
            'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity', 'Hue',
            'OD280/OD315 of diluted wines', 'Proline']
    scaler = StandardScaler()
    scaled_data = scaler.fit_transform(train_df[cols[1:]])
    scaled_df = pd.DataFrame(scaled_data, columns=cols[1:])
    scaled_df.insert(0, 'class', train_df['class'])

    scaled_df.to_csv('data_use.csv', index=False)
```

(6) 数据的相关性分析

```
    import seaborn as sns
    correlation_matrix = scaled_df[cols[1:]].corr()

    plt.figure(figsize=(10, 8))
    sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
    plt.title('Correlation Heatmap')
    plt.show()
```

(7) PCA 降维

```
    from sklearn.decomposition import PCA

    # 获取除了class之外的所有数据列
    data_2 = scaled_df[cols[1:]].values

    # 创建PCA对象并拟合数据
    pca = PCA(n_components=2)
    pca.fit(data_2)

```

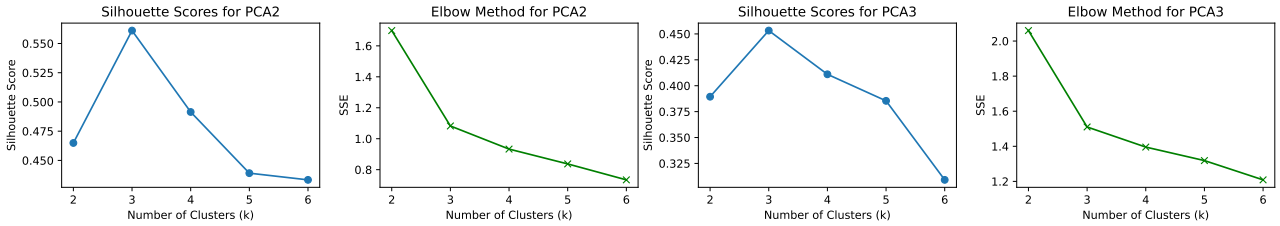
```
# 转换数据到低维空间
transformed_data_2 = pca.transform(data_2)

# 将降维后的数据转换为DataFrame
df_transformed_2 = pd.DataFrame(transformed_data_2, columns=['PC1', 'PC2'])

# 将'class'列与降维后的数据合并
df_combined_2 = pd.concat([scaled_df['class'], df_transformed_2], axis=1)

# 将合并后的数据保存到CSV文件
df_combined_2.to_csv('PCA_2.csv', index=False)
```

3. 聚类模型搭建

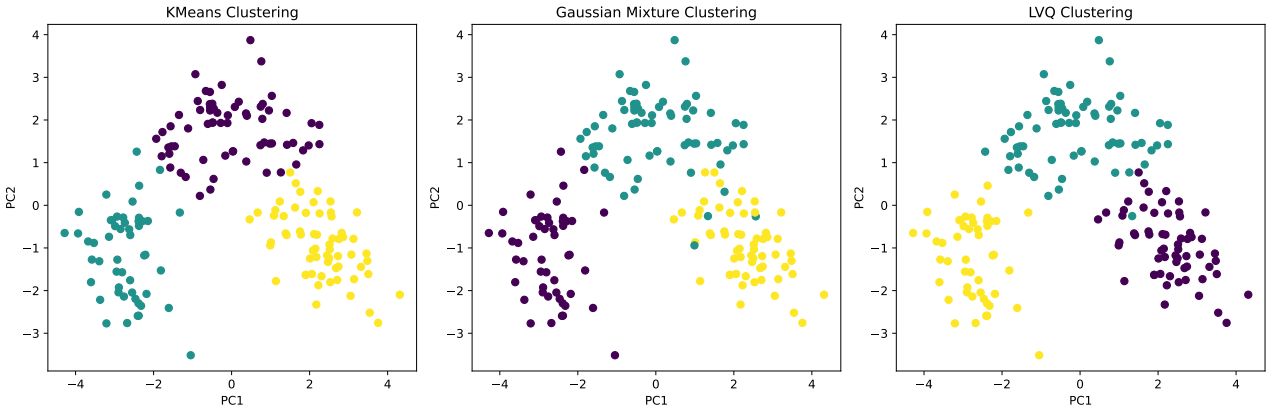
(1) 不同聚类方法的结果分析

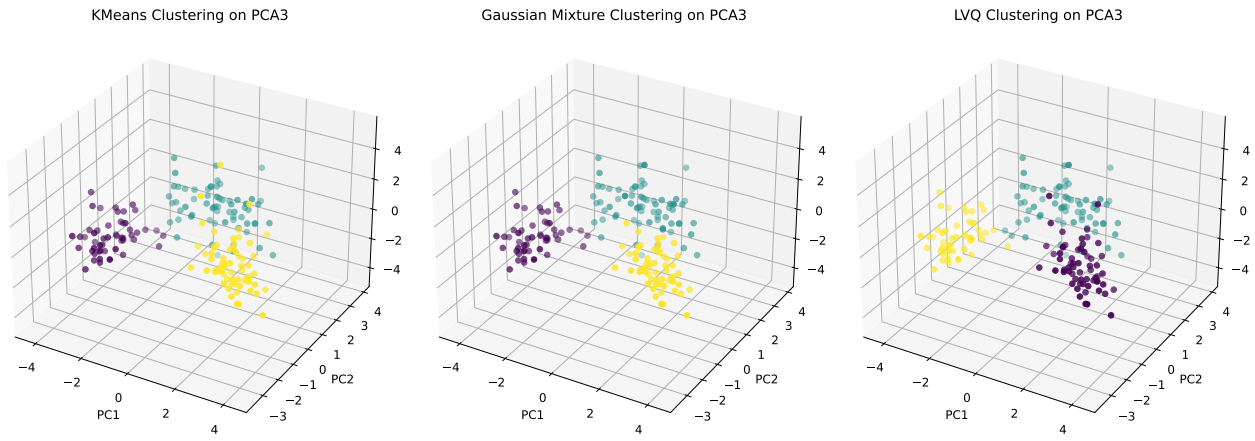
我们在实验中分别对 PCA2 与 PCA3 使用了k_means，高斯混合聚类，学习向量量化进行聚类，并对聚类结果进行了可视化并计算了各自的 FMI 系数。

在聚类分析中，确定最优的簇（群集）数量是一个关键步骤。轮廓系数法（*Silhouette Method*）和肘部法（*Elbow Method*）是两种估计最佳簇数量的常用方法。我们在实验中对于 PCA2 与 PCA3 分别使用了这两种方法来对聚类数量进行估计，实验的可视化结果如下：

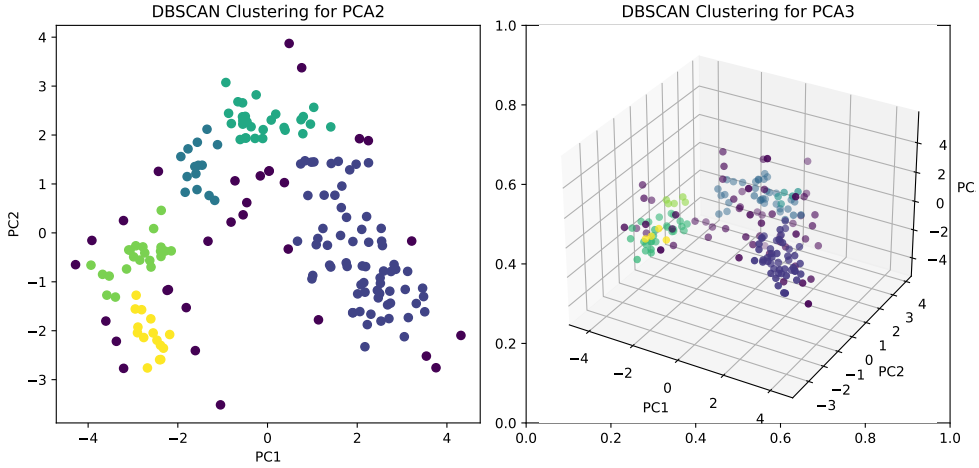


接下来我们通过调用模型库的方式分别对 PCA2 与 PCA3 使用三种方式进行聚类，实验可视化结果如下：





实验中我们也尝试了一下密度聚类，效果不是很好：



选取不同主成分和不同方法的 FMI 系数计算结果如表1 所示，从表格中可以看出在本数据集上聚类效果：学习向量 > 高斯聚类 >=k_means> 密度聚类学习向量效果较好推断主要是由于聚类中利用率到了类别标签。

聚类方法	PCA2 FMI 分数	PCA3 FMI 分数
K-Means	0.9304	0.9205
高斯混合聚类	0.9427	0.9207
学习向量聚类	0.9417	0.9649
密度聚类	0.5582	0.5942

表 1: 不同聚类方法和 PCA 维数下的 FMI 分数比较

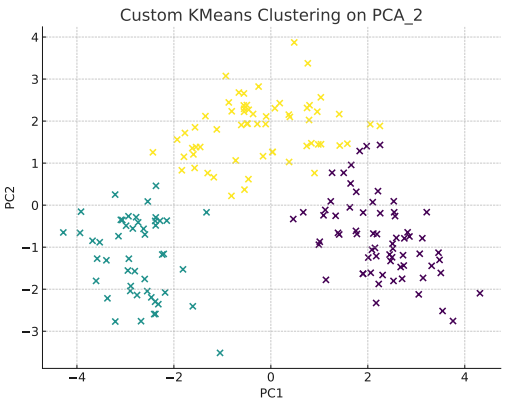
(2) kmeans 聚类的手动实现

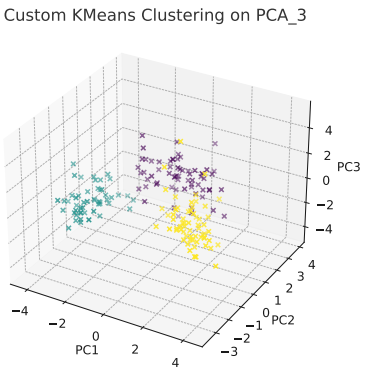
```python
class CustomKMeans:
    def __init__(self, n_clusters=3, max_iter=300, tol=1e-4):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.tol = tol
        self.centroids = None

    def fit(self, data):
        # Initialize centroids by randomly selecting data points
        self.centroids = data[np.random.choice(data.shape[0], self.n_clusters, replace=False)]

        for i in range(self.max_iter):
            # Assign clusters based on closest centroid
            distances = np.sqrt(((data - self.centroids[:, np.newaxis])**2).sum(axis=2))
            self.labels_ = np.argmin(distances, axis=0)

            # Calculate new centroids
            new_centroids = np.array([data[self.labels_ == j].mean(axis=0) for j in range(self.n_clusters)])

            # Check for convergence
            if np.all(np.abs(new_centroids - self.centroids) <= self.tol):
                break

            self.centroids = new_centroids

    def predict(self, data):
        distances = np.sqrt(((data - self.centroids[:, np.newaxis])**2).sum(axis=2))
        return np.argmin(distances, axis=0)
```

这个 CustomKMeans 类是一个简化的 K 均值聚类算法实现，能够根据设定的簇数量、迭代次数和收敛阈值对数据进行聚类。它具有以下方法和属性：

1. `__init__(self, n_clusters=3, max_iter=300, tol=1e-4)`：构造函数，用于初始化类的实例。它接收三个参数：n_clusters定义了要形成的簇的数量，默认为 3；max_iter定义了算法的最大迭代次数，默认为 300；tol定义了收敛阈值，如果新的中心点与旧的中心点之间的变化小于这个阈值，则认为已经收敛，默认值为 0.0001。
2. `fit(self, data)`：这个方法接收数据并对其进行聚类。它首先随机选择数据点作为初始质心，然后迭代地执行以下步骤：计算每个数据点到每个质心的距离，为每个数据点分配最近的质心作为其簇，然后根据簇内的数据点重新计算质心。如果新质心与旧质心的差异小于 tol，或者达到 max_iter 次迭代，则停止迭代。
3. `predict(self, data)`：该方法用于将数据点分配到最近的质心形成的簇。它计算了数据点到每个质心的距离，并返回最近质心的索引，即每个数据点所属的簇。

最终可视化出来的结果如下，整体来看和效果和调库实现的结果十分相似。





(3) 层次聚类的手动实现

距离计算函数

这部分代码定义了三种不同的距离计算方法：欧式距离、曼哈顿距离和切比雪夫距离。

1. 欧式距离（euclidean_distance）：这是最常用的距离度量方法，计算两点之间的直线距离。
2. 曼哈顿距离（manhattan_distance）：这种距离度量计算的是在标准坐标系统上两点间的绝对轴距总和，类似于在城市街道上从一点到另一点的距离（考虑街道的布局）。
3. 切比雪夫距离（chebyshev_distance）：这种度量方法计算的是两点之间各坐标数值差的最大值，适用于需要考虑极端差异的情况。

```python
# 计算欧式距离
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((np.array(point1) - np.array(point2)) ** 2))

# 计算曼哈顿距离
def manhattan_distance(point1, point2):
    return np.sum(np.abs(np.array(point1) - np.array(point2)))

# 计算切比雪夫距离
def chebyshev_distance(point1, point2):
    return np.max(np.abs(np.array(point1) - np.array(point2)))
```

四种类间距函数

这部分代码定义了四种不同的类间距离计算方法，用于层次聚类中确定簇间的距离。

1. 单链聚类（single_linkage）：计算两个簇中距离最近的点对之间的距离。适用于发现非球形簇。
2. 完全链接聚类（complete_linkage）：计算两个簇中距离最远的点对之间的距离。倾向于形成大小相近的簇。

3. 组平均聚类（average_linkage）：计算两个簇中所有点对距离的平均值。是一种折中的距离计算方式，平衡了单链聚类和完全链接聚类的极端。
4. 距离中心点聚类（centroid_linkage）：先计算每个簇的中心点，然后计算两个簇中心点之间的距离。这种方法关注簇的整体中心位置。

```python
# 单链聚类的距离计算
def single_linkage(cluster1, cluster2, data):
    min_distance = float('inf')
    for index1 in cluster1:
        for index2 in cluster2:
            distance = calculate_distance(data[index1], data[index2])
            if distance < min_distance:
                min_distance = distance
    return min_distance

# 完全链接聚类的距离计算
def complete_linkage(cluster1, cluster2, data):
    max_distance = float('-inf')
    for index1 in cluster1:
        for index2 in cluster2:
            distance = calculate_distance(data[index1], data[index2])
            if distance > max_distance:
                max_distance = distance
    return max_distance

# 组平均聚类的距离计算
def average_linkage(cluster1, cluster2, data):
    total_distance = 0
    num_pairs = 0
    for index1 in cluster1:
        for index2 in cluster2:
            distance = calculate_distance(data[index1], data[index2])
            total_distance += distance
            num_pairs += 1
    return total_distance / num_pairs if num_pairs > 0 else 0

# 计算聚类的中心点
def calculate_centroid(cluster_indices, data):
    points=[]
    for it in np.array(cluster_indices):
        points.append(data[it])
    centroid = np.mean(points, axis=0)
    return centroid
# 距离中心点聚类的距离计算
def centroid_linkage(cluster1, cluster2, data):
    # 计算每个聚类的中心点
    centroid1 = calculate_centroid(cluster1, data)
    centroid2 = calculate_centroid(cluster2, data)
    # 计算两个中心点之间的距离
    return calculate_distance(centroid1, centroid2)
```

凝聚层次聚类实现

这部分代码是层次聚类算法的具体实现，使用凝聚的方式。算法从将每个数据点视为一个单独的簇开始，然后逐步合并最接近的簇，直到达到指定的簇数量。

- 算法首先初始化所有数据点为单独的簇。
- 然后，算法重复寻找最近的两个簇并将它们合并，直到只剩下指定数量的簇。
- 在每次迭代中，可以根据单链聚类、完全链接聚类、组平均聚类或距离中心点聚类的方法来计算簇间距离。

```
# 层次聚类算法
def hierarchical_clustering(data, num_clusters):
    clusters = [[i] for i in range(len(data))]
    while len(clusters) > num_clusters:
        min_distance = float('inf')
        clusters_to_merge = (0, 1)
        for i in range(len(clusters)):
            for j in range(i+1, len(clusters)):
                #几种不同的类间距离衡量
                #distance = single_linkage(clusters[i], clusters[j], data)      #MIN
                #distance = complete_linkage(clusters[i], clusters[j], data)    #MAX
                distance = average_linkage(clusters[i], clusters[j], data)     #Group Average
                #distance = centroid_linkage(clusters[i], clusters[j], data)    #Distance Between Centroids
                if distance < min_distance:
                    min_distance = distance
                    clusters_to_merge = (i, j)
        # 合并最近的两个簇
        clusters[clusters_to_merge[0]].extend(clusters[clusters_to_merge[1]])
        del clusters[clusters_to_merge[1]]
    return clusters

clusters = hierarchical_clustering(data_for_clustering.tolist(), 3)
```

FMI 计算

这部分代码用于计算 Fowlkes-Mallows 指数（FMI），这是一种评估聚类质量的指标，特别是在有真实标签可用时。

- 首先，代码根据层次聚类算法的结果为每个数据点分配一个簇标签。
- 然后，它计算真实标签和聚类结果之间的一致性。
  - TP（真阳性）：同一个簇且真实标签相同的点对数量。
  - FP（假阳性）：同一个簇但真实标签不同的点对数量。
  - TN（真阴性）：不同簇且真实标签不同的点对数量。
  - FN（假阴性）：不同簇但真实标签相同的点对数量。
- FMI 计算为 $\frac{TP}{\sqrt{((TP+FP)*(TP+FN))}}$，值越高表示聚类效果越好。

```python
# 计算Fowlkes-Mallows指数
original_labels = mydata_array[:, 0]
cluster_labels = np.zeros_like(original_labels)
for cluster_id, cluster in enumerate(clusters):
    for index in cluster:
        cluster_labels[index] = cluster_id
TP = FP = FN = TN = 0
for i, j in combinations(range(len(cluster_labels)), 2):
    same_cluster_original = (original_labels[i] == original_labels[j])
    same_cluster_new = (cluster_labels[i] == cluster_labels[j])
    if same_cluster_original and same_cluster_new:
        TP += 1
    elif not same_cluster_original and not same_cluster_new:
        TN += 1
    elif not same_cluster_original and same_cluster_new:
        FP += 1
    elif same_cluster_original and not same_cluster_new:
        FN += 1
FMI = TP / np.sqrt((TP + FP) * (TP + FN))
```

分裂层次聚类

这部分代码提供了分裂层次聚类的实现，这是层次聚类的另一种形式，从一个包含所有数据点的单一簇开始，逐步细分为更小的簇。

- 算法开始时，所有数据点被视为一个大簇。
- 使用 K 均值聚类（KMeans）将这个大簇分裂成指定数量的子簇。
- 然后，每个子簇被视为独立的簇，并返回最终的簇列表。

这种方法与凝聚层次聚类相反，从一个大簇开始逐步分裂，而不是从许多小簇开始逐步合并。这种方法适用于数据集较大或簇的结构较为复杂时。

```python
def divisive_clustering(data, num_clusters):
    clusters = [np.arange(data.shape[0])]
    cluster_to_split = clusters[0]
    kmeans = KMeans(n_clusters=num_clusters,init='k-means++').fit(data[cluster_to_split])
    new_clusters_labels = kmeans.labels_
    clusters=[]
    for it in range(num_clusters):
        clusters.append(cluster_to_split[new_clusters_labels == it])
    return clusters
```

结果可视化