《大数据分析与挖掘》
实验报告

姓　名：刘军　　学　号：20212753

学　院：计算机与数学学院　　年　级：2021　级

专业班级：计算机科学与技术三班　　指导老师：周健老师

报告评语：

成绩：

目录

课设一：基于 Apriori 算法的关联规则算例

一、实验目的

1、了解 Apriori 算法及其关联规则的实例
2、使用 Python 根据书中的示例实现 Apriori 算法

二、实验任务

某超市有五条客户购物清单记录，设定最小支持度为 40%，最小置信度为 60%，使用 Apriori 算法计算频繁项集和关联规则。

三、实验过程

为了通过 Python 实现 Apriori 算法并找出频繁项集和关联规则，我们可以借助 mlxtend 库提供的方便 Apriori 功能。在 association_rules 函数中，metric 参数用于确定计算规则质量的度量，而 min_threshold 参数用于设定此度量的最小阈值。
购物清单
['咖啡', '面包', '香肠', '果酱', '香皂']
['果酱', '香肠']
['咖啡', '洗衣粉', '香肠']
['咖啡', '面包', '牛奶']
['咖啡', '香肠', '牛奶']
实验代码

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules

# 购物清单记录
dataset = [['咖啡', '面包', '香肠', '果酱', '香皂'],
        ['果酱', '香肠'],
        ['咖啡', '洗衣粉', '香肠'],
        ['咖啡', '面包', '牛奶'],
        ['咖啡', '香肠', '牛奶']]

# 使用 TransactionEncoder 将数据集转换为适合 Apriori 算法的形式
te = TransactionEncoder()
te_ary = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_ary, columns=te.columns_)
```

```
# 设置最小支持度和最小置信度
min_support = 0.4  # 40%
min_confidence = 0.6  # 60%

# 使用 apriori 函数找出频繁项集
frequent_itemsets = apriori(df, min_support=min_support, use_colnames=True)

# 使用 association_rules 函数从频繁项集中生成关联规则
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=min_confidence)

# 打印频繁项集
print("Frequent Itemsets:")
print(frequent_itemsets)

# 打印关联规则
print("\nAssociation Rules:")
print(rules[['antecedents', 'consequents', 'support', 'confidence']])
```

四、实验结果

```
D:\Users\LJ189\PycharmProjects\keshe\.venv\Scripts\python.exe D:\Users\LJ189\PycharmProjects\keshe\1.py
Frequent Itemsets:
   support  itemsets
0      0.8      (咖啡)
1      0.4      (果酱)
2      0.4      (牛奶)
3      0.4      (面包)
4      0.8      (香肠)
5      0.4  (咖啡, 牛奶)
6      0.4  (咖啡, 面包)
7      0.6  (咖啡, 香肠)
8      0.4  (香肠, 果酱)

Association Rules:
  antecedents consequents  support  confidence
0        (牛奶)        (咖啡)      0.4        1.00
1        (面包)        (咖啡)      0.4        1.00
2        (咖啡)        (香肠)      0.6        0.75
3        (香肠)        (咖啡)      0.6        0.75
4        (果酱)        (香肠)      0.4        1.00
```

五、个人总结

本实验旨在深入了解并应用 Apriori 算法，以探索购物篮分析在超市购物数据中的应用。Apriori 算法是一种经典的关联规则挖掘方法，适用于发现数据集中的频繁项集和关联规则。通过 Python 实现 Apriori 算法，并结合 mlxtend 库提供的功能，我们能够在超市购物清单数据中找出频繁项集和关联规则。该算法通过设置最小支持度和最小置信度等参数来过滤出具有实际意义的结果，从而帮助我们理解客户购物行为和产品之间的关联关系。

课设二：Apriori 关联规则方法的实例

一、实验目的：

1. 掌握 Apriori 算法的基本框架
2. 使用 Python 编写 Apriori 关联规则的示例代码

二、实验任务：

了解高等教育与性别、工资收入、职业、年龄等之间的潜在关联，并使用关联规则分析这些关系。提供一个简单的数据库示例，并将其转换为布尔值表。

三、实验过程：

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules

# 部分 1：将分类数据转换为逻辑值
df = pd.DataFrame({
   'SEX': [1, 2, 1, 1, 1, 1],  # 1:male, 2:female
   'AGE': [3, 4, 4, 3, 3, 3],  # 3:old, 4:young
   'KNOWLEDGE': [5, 5, 6, 5, 5, 6],  # 5:high, 6:low
   'OCCUPATION': [7, 7, 8, 7, 7, 8],  # 7:Teacher, 8:Technician
   'WAGES': [9, 9, 10, 9, 9, 10]  # 9:high, 10:low
})
df_one_hot = pd.get_dummies(df, columns=['SEX', 'AGE', 'KNOWLEDGE', 'OCCUPATION', 'WAGES'])
pd.set_option('display.max_columns', None)
print(df_one_hot)

# 部分 2：设最小支持度为 0.5,最小置信度为 0.7 求得关联规则
data = [
   {'RECID': 100, 'SEX': 'male', 'AGE': 46, 'KNOWLEDGE': 'Doctor', 'OCCUPATION': 'Teacher',
'WAGES': 7500},
   {'RECID': 200, 'SEX': 'female', 'AGE': 32, 'KNOWLEDGE': 'Master', 'OCCUPATION': 'Teacher',
'WAGES': 6500},
   {'RECID': 300, 'SEX': 'male', 'AGE': 35, 'KNOWLEDGE': 'Bachelor', 'OCCUPATION': 'Technician',
'WAGES': 4900},
   {'RECID': 400, 'SEX': 'male', 'AGE': 40, 'KNOWLEDGE': 'Master', 'OCCUPATION': 'Teacher',
'WAGES': 6000},
```

```
  {'RECID': 500, 'SEX': 'male', 'AGE': 37, 'KNOWLEDGE': 'Doctor', 'OCCUPATION': 'Teacher', 'WAGES': 7000},
  {'RECID': 600, 'SEX': 'male', 'AGE': 25, 'KNOWLEDGE': 'Bachelor', 'OCCUPATION': 'Technician', 'WAGES': 4000},
]

df = pd.DataFrame(data)
df['SEX'] = df['SEX'].map({'male': 1, 'female': 2})
df['AGE'] = df['AGE'].apply(lambda x: 3 if x > 40 else 4)
df['KNOWLEDGE'] = df['KNOWLEDGE'].apply(lambda x: 5 if x in ['Doctor', 'Master'] else 6)
df['OCCUPATION'] = df['OCCUPATION'].apply(lambda x: 7 if x == 'Teacher' else 8)
df['WAGES'] = df['WAGES'].apply(lambda x: 9 if x > 5000 else 10)

te = TransactionEncoder()
te_ary = te.fit(df[['SEX', 'AGE', 'KNOWLEDGE', 'OCCUPATION', 'WAGES']]).transform(df[['SEX', 'AGE', 'KNOWLEDGE', 'OCCUPATION', 'WAGES']])
df_transactions = pd.DataFrame(te_ary, columns=te.columns_)

frequent_itemsets = apriori(df_transactions, min_support=0.5, use_colnames=True)
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1)
rules_with_high_confidence = rules[rules['confidence'] >= 0.7]
print(rules_with_high_confidence[['antecedents', 'consequents', 'support', 'confidence', 'lift']])
```

四、实验结果：

```
D:\Users\LJ189\PycharmProjects\keshe\.venv\Scripts\python.exe D:\Users\LJ189\PycharmProjects\keshe\2.py
   SEX_1  SEX_2  AGE_3  AGE_4  KNOWLEDGE_5  KNOWLEDGE_6  OCCUPATION_7  \
0   True  False   True  False         True        False          True
1  False   True  False   True         True        False          True
2   True  False  False   True        False         True         False
3   True  False   True  False         True        False          True
4   True  False   True  False         True        False          True
5   True  False   True  False        False         True         False

   OCCUPATION_8  WAGES_9  WAGES_10
0         False     True     False
1         False     True     False
2          True    False      True
3         False     True     False
4         False     True     False
5          True    False      True
  antecedents consequents  support  confidence  lift
0         (G)         (E)      0.5        1.00   1.5
1         (E)         (G)      0.5        0.75   1.5
```

五、个人总结：

在本次实验中，我们深入了解了 Apriori 算法的基本框架，并成功使用 Python 编写了生成关联规则的示例代码。通过对关联规则的分析，我们能够发现数据集中隐藏的有趣模式和关系。具体来说，我们理解了如何使用支持度和置信度来衡量关联规则的强度，以及如何通过剪枝过程提高算法的效率。本次实验不仅强化了我们对 Apriori 算法的理解，还增强了我们使用 Python 进行数据挖掘的能力。

课设三：k-means 聚类方法

一、实验目的：

1、了解不同种类的聚类方法。
2、熟悉 k-means 算法的核心思想。

二、实验任务：

给定一个数据集 X，包含样本 {1}，{5, 10, 9}，和 {26, 32, 16, 21, 14}。我们的目标是将这些样本聚类成 3 个簇（k=3）。首先，我们随机选择了前三个数值作为初始的聚类中心：z1=1，z2=5，z3=10（使用欧氏距离进行计算）。
在第一次迭代中，我们根据这三个初始聚类中心将样本集合分为三个簇：{1}，{5}，和 {10, 9, 26, 32, 16, 21, 14}。然后，针对每个产生的簇，我们计算了其平均值点，得到了 1，5，和 18.3。接着，我们将这些平均值点填入第一步中的 z1, z2, z3 栏中。
在第二次迭代中，我们通过计算新的平均值点重新调整了样本所在的簇。即，我们根据离每个样本最近的平均值点（1，5，18.3）的原则，将样本重新分配到簇中。这个过程得到了三个新的簇：{1}，{5, 10, 9}，和 {26, 32, 16, 21, 14}，并将它们填入了第二步的 C1，C2，和 C3 栏中。随后，我们重新计算了每个簇的平均值点，得到了新的平均值点 1，8，和 21.8。这个迭代过程持续进行，直到第五次迭代时，得到的簇与第四次迭代的结果相同，同时目标函数 E 收敛，迭代结束。

三、实验过程：

```
import numpy as np

def euclidean_distance(point, centers):
    return np.sqrt(np.sum((point - centers)**2, axis=1))

def kmeans(X, k, iterations=100):
    # 初始化聚类中心为随机选取的 k 个样本点
    centers = X[np.random.choice(len(X), k, replace=False)]

    for iter_num in range(iterations):
        # 计算每个样本点到各个聚类中心的距离
        distances = np.array([euclidean_distance(x, centers) for x in X])
        # 将样本点分配给距离最近的聚类中心
        labels = np.argmin(distances, axis=1)

        # 更新聚类中心为每个簇的均值
        new_centers = np.array([X[labels == i].mean(axis=0) for i in range(k)])
```

```
        # 输出每次迭代的结果
        print(f"第 {iter_num+1} 次迭代:")
        for i, center in enumerate(centers):
            print(f"簇 {i+1} 中心: {center[0]:.2f}")
        print("标签:", labels)
        print()

        # 如果聚类中心不再变化，则退出迭代
        if np.all(centers == new_centers):
            break

        centers = new_centers

    return centers, labels

# 测试
X = np.concatenate(([1], [5, 10, 9], [26, 32, 16, 21, 14])).astype(float)
k = 3
centers, labels = kmeans(X.reshape(-1, 1), k)

print("最终的聚类中心:")
for i, center in enumerate(centers):
    print(f"簇 {i+1} 中心: {center[0]:.2f}")

print("\n 样本点的簇分配:")
for i, label in enumerate(labels):
    print(f"样本点 {i+1}: 属于簇 {label+1}")
```

四、实验结果：

```
D:\Users\LJ189\PycharmProjects\keshe\.venv\Scripts\python.exe D:\Users\LJ189\PycharmProjects\keshe\3.py
第 1 次迭代:
簇 1 中心: 1.00
簇 2 中心: 9.00
簇 3 中心: 32.00
标签: [0 0 1 1 2 2 1 2 1]

第 2 次迭代:
簇 1 中心: 3.00
簇 2 中心: 12.25
簇 3 中心: 26.33
标签: [0 0 1 1 2 2 1 2 1]

最终的聚类中心:
簇 1 中心: 3.00
簇 2 中心: 12.25
簇 3 中心: 26.33

样本点的簇分配:
样本点 1: 属于簇 1
样本点 2: 属于簇 1
样本点 3: 属于簇 2
样本点 4: 属于簇 2
样本点 5: 属于簇 3
样本点 6: 属于簇 3
样本点 7: 属于簇 2
样本点 8: 属于簇 3
```

五、个人总结：

通过本次实验，我深入了解了不同类型的聚类方法，并着重学习了 k-means 算法的核心原理。在实践中，掌握了如何使用 k-means 算法将给定的数据集聚类成预先设定的簇数量。这个过程中，学会了如何使用欧式距离来衡量样本之间的相似性，并通过迭代过程不断调整簇的中心点，直到算法收敛为止。

通过这次实验，加深了对聚类算法的理解，学会了将理论知识应用到实际问题中。同时，也意识到了在实际应用中，选择合适的簇数量和初始中心点对聚类结果具有重要影响。这些经验将对未来在数据分析和机器学习领域的工作产生积极影响。

课设四：k-medoids 聚类方法

一、实验目的：

1、通过实践，理解 k-medoids 算法的基本原理和核心思想。
2、掌握使用 Python 编写代码实现 k-medoids 算法的方法，从而能够运用该算法解决实际问题。

二、实验任务：

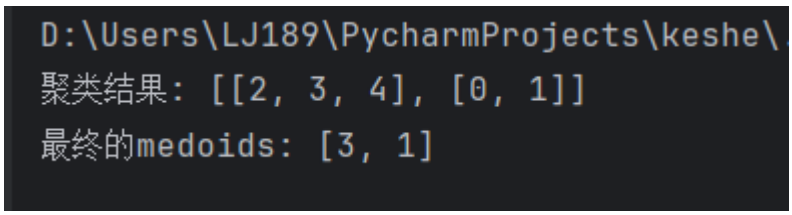
假如空间中的五个点(A，B，C,D,E},各点之间的距离关系如书上表 6.2 所示

三、实验过程：

```
import numpy as np
from pyclustering.cluster.kmedoids import kmedoids
from pyclustering.utils.metric import distance_metric, type_metric

# 样本点之间的距离矩阵
distance_matrix = np.array([
    [0, 1, 2, 2, 3],
    [1, 0, 2, 4, 3],
    [2, 2, 0, 1, 5],
    [2, 4, 1, 0, 3],
    [3, 3, 5, 3, 0]
])

# 初始 medoids
initial_medoids = [0, 1]

# 创建自定义距离度量
metric = distance_metric(type_metric.USER_DEFINED, func=lambda x, y: distance_matrix[int(x)][int(y)])

# 创建 K-Medoids 实例
kmedoids_instance = kmedoids(data=list(range(len(distance_matrix))), initial_index_medoids=initial_medoids, metric=metric)

# 运行聚类分析
kmedoids_instance.process()
```

```
# 获取聚类结果
clusters = kmedoids_instance.get_clusters()
medoids = kmedoids_instance.get_medoids()

# 打印聚类结果和最终的 medoids
print(f'聚类结果: {clusters}')
print(f'最终的 medoids: {medoids}')
```

四、实验结果：



五、个人总结：

通过实践，我们能够深入理解 k-medoids 算法的基本原理和核心思想。该算法的核心思想是选择代表性的点作为簇的中心，而不是简单地使用均值。这些代表性点被称为 medoids。k-medoids 算法通过迭代的方式，不断更新 medoids 以最小化簇内的总距离，从而实现聚类过程。

掌握使用 Python 编写代码实现 k-medoids 算法的方法是关键的一步。通过编写代码，我们可以直观地理解算法的运行过程，并在实际问题中应用该算法。在编写代码时，需要考虑如何计算距离矩阵、选择初始 medoids、更新 medoids 等步骤，以确保算法能够正确地聚类数据集。

课设五：AGNES 聚类方法

一、实验目的：

1、理解 AGNES（Agglomerative Nesting）聚类算法的基本原理，即自下而上的聚合式聚类方法。
2、使用 Python 编写代码实现 AGNES 算法，并运用该算法解决书中提供的实例问题，以加深对算法原理的理解和应用。

二、实验任务：

下面给出一个样本事物数据库,如表 6.3 所示,并对它实施 AGNES 算法序号 属性 1 属性 2 1 1 1 2 1 2 3 2 1 4 2 2 5 3 4 6 3 5 7 4 4 8 4 5,请你使用 python 实现该问题

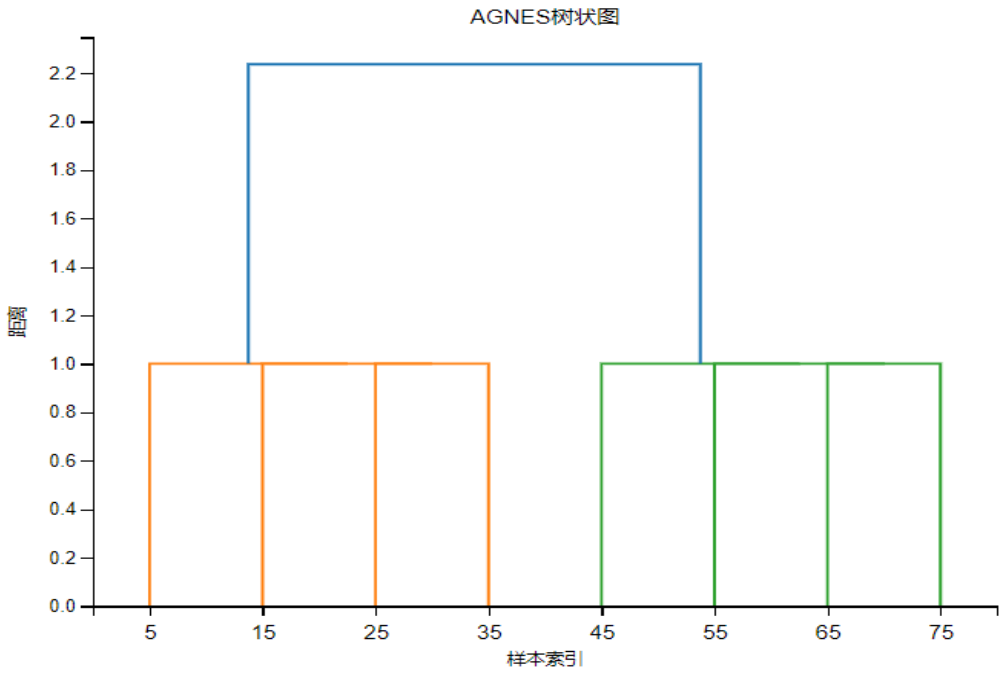
三、实验过程：

```
import numpy as np
import pandas as pd
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# 数据
data = np.array([
    [1, 1],
    [1, 2],
    [2, 1],
    [2, 2],
    [3, 4],
    [3, 5],
    [4, 4],
    [4, 5]
])

# 将数据转换为 DataFrame
df = pd.DataFrame(data, columns=['属性 1', '属性 2'])

# 使用 scipy 的 linkage 函数计算层次聚类
Z = linkage(df, method='single', metric='euclidean')

# 输出每次合并后的簇
print("链接矩阵:\n", Z)
```

```
# 绘制树状图
plt.figure(figsize=(10, 7))
plt.title("AGNES 树状图")
dendrogram(Z)
plt.xlabel('样本索引')
plt.ylabel('距离')
plt.rcParams['font.sans-serif'] = ['SimHei']  # 设置中文字体为黑体
plt.show()
```

四、实验结果：

```
D:\Users\LJ189\PycharmProjects\keshe\.venv\Scripts\python.exe D:
链接矩阵:
 [[ 0.          1.          1.          2.        ]
 [ 2.          8.          1.          3.        ]
 [ 3.          9.          1.          4.        ]
 [ 4.          5.          1.          2.        ]
 [ 6.         11.          1.          3.        ]
 [ 7.         12.          1.          4.        ]
 [10.         13.          2.23606798  8.        ]]
```



五、个人总结：

通过本次实验，我们学习了 AGNES（Agglomerative Nesting）算法的基本原理和实现方法。AGNES 算法是一种自下而上的层次聚类算法，通过逐步合并最近的样本来构建聚类。在实验中，我们首先将样本数据转换为 DataFrame 格式，并使用 Scipy 的 linkage 函数计算了层次聚类。然后，我们通过绘制树状图展示了聚类结果，树状图清晰地展示了样本之间的聚类关系。

通过本次实验，我们加深了对 AGNES 算法的理解，掌握了如何使用 Python 实现该算法，并对层次聚类的原理有了更深入的认识。这些知识和经验将有助于我们在实际问题中应用聚类算法，并对数据进行更深入的分析和理解。

课设六：DBSCAN 聚类方法

一、实验目的：

1、理解 DBSCAN（Density-Based Spatial Clustering of Applications with Noise）聚类算法的基本原理，即基于密度的空间聚类算法。
2、使用 Python 编写代码，并运用 DBSCAN 算法解决书上给出的实例问题，以加深对算法原理的理解和应用。

二、实验任务：

下面给出一个样本事务数据库,如下所示,并对它实施 DBSCAN 算法。
序号:1,2,3,4,5,6,7,8,9,10,11,12;属性 1:1,4,0,1,2,3,4,5,0,1,4,1;属性 2:0,0,1,1,1,1,1,1,2,2,2,3;
设 n=12,e=1,MinPts=4,

三、实验过程：

```
import numpy as np

# 示例数据集
data = np.array([
    [1, 0],
    [4, 0],
    [0, 1],
    [1, 1],
    [2, 1],
    [3, 1],
    [4, 1],
    [5, 1],
    [0, 2],
    [1, 2],
    [4, 2],
    [1, 3]
])

# 定义函数：计算两点之间的欧式距离
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))

# 定义函数：找到给定点的 eps 邻域内的所有点
def find_neighbors(point_index, data, eps):
```

```
    point = data[point_index]
    return [i for i, p in enumerate(data) if i != point_index and euclidean_distance(p, point) <= eps]

# 定义函数：DBSCAN 算法
def dbscan(data, eps, min_samples):
    n_samples = len(data)
    labels = [-1] * n_samples  # 初始化所有点的标签为-1（表示噪声或未访问）
    cluster_id = 0  # 簇的 ID 计数器
    visited = set()  # 存储已访问的点的索引

    for point_index in range(n_samples):
        if point_index in visited:  # 如果这个点已经被访问过，则跳过
            continue

        visited.add(point_index)  # 标记当前点为已访问
        point = data[point_index]  # 获取当前点的坐标
        neighbors = find_neighbors(point_index, data, eps)  # 查找 eps 内的邻居

        n_neighbors = len(neighbors)  # eps 内邻居点的个数

        if n_neighbors < min_samples:  # 如果邻居点不足，标记为噪声
            labels[point_index] = -1
            print(f"步骤 1: 点 {point} 被标记为噪声，因为其 eps 邻域内的点个数为 {n_neighbors}（小于 {min_samples}）。")
        else:
            # 开始一个新的簇
            cluster_id += 1
            labels[point_index] = cluster_id  # 给当前点分配簇标签

            # 扩展簇
            expand_cluster(data, labels, point_index, neighbors, cluster_id, eps, min_samples, visited)

    return labels

# 定义函数：扩展簇
def expand_cluster(data, labels, point_index, neighbors, cluster_id, eps, min_samples, visited):
    queue = neighbors.copy()
    step = 1
    while queue:
        current_point_index = queue.pop(0)
        if labels[current_point_index] == -1:  # 如果邻居点是噪声，则标记为当前簇的成员
            labels[current_point_index] = cluster_id
        elif labels[current_point_index] == 0 and current_point_index not in visited:  # 如果邻居点还未被标记
```

```
            labels[current_point_index] = cluster_id
            current_point_neighbors = find_neighbors(current_point_index, data, eps)
            if len(current_point_neighbors) >= min_samples:
                queue.extend(current_point_neighbors)
        visited.add(current_point_index)
        step += 1
        print(f"步骤 {step}: 开始一个新簇（ID：{cluster_id}），选择的点：{data[current_point_index]}")
        print(f"    在 eps 内的点的个数：{len(neighbors)}")
        print(f"    通过计算可达点而找到的新簇成员：{[data[i] for i in queue]}")

# 调用 DBSCAN 算法并输出结果
eps = 1
min_samples = 4
labels = dbscan(data, eps, min_samples)
print("聚类结果:", labels)
```

四、实验结果：

```
D:\Users\LJ189\PycharmProjects\keshe\.venv\Scripts\python.exe D:\Users\LJ189\Pyc
步骤 1: 点 [1 0] 被标记为噪声，因为其 eps 邻域内的点个数为 1（小于 4）。
步骤 1: 点 [4 0] 被标记为噪声，因为其 eps 邻域内的点个数为 1（小于 4）。
步骤 1: 点 [0 1] 被标记为噪声，因为其 eps 邻域内的点个数为 2（小于 4）。
步骤 2: 开始一个新簇（ID：1），选择的点：[1 0]
    在 eps 内的点的个数：4
    通过计算可达点而找到的新簇成员：[array([0, 1]), array([2, 1]), array([1, 2])]
步骤 3: 开始一个新簇（ID：1），选择的点：[0 1]
    在 eps 内的点的个数：4
    通过计算可达点而找到的新簇成员：[array([2, 1]), array([1, 2])]
步骤 4: 开始一个新簇（ID：1），选择的点：[2 1]
    在 eps 内的点的个数：4
    通过计算可达点而找到的新簇成员：[array([1, 2])]
步骤 5: 开始一个新簇（ID：1），选择的点：[1 2]
    在 eps 内的点的个数：4
    通过计算可达点而找到的新簇成员：[]
步骤 1: 点 [3 1] 被标记为噪声，因为其 eps 邻域内的点个数为 2（小于 4）。
步骤 2: 开始一个新簇（ID：2），选择的点：[4 0]
    在 eps 内的点的个数：4
    通过计算可达点而找到的新簇成员：[array([3, 1]), array([5, 1]), array([4, 2])]
步骤 3: 开始一个新簇（ID：2），选择的点：[3 1]
    在 eps 内的点的个数：4
    通过计算可达点而找到的新簇成员：[array([5, 1]), array([4, 2])]
步骤 4: 开始一个新簇（ID：2），选择的点：[5 1]
    在 eps 内的点的个数：4
    通过计算可达点而找到的新簇成员：[array([4, 2])]
步骤 5: 开始一个新簇（ID：2），选择的点：[4 2]
    在 eps 内的点的个数：4
    通过计算可达点而找到的新簇成员：[]
```

```
步骤 1: 点 [0 2] 被标记为噪声，因为其 eps 邻域内的点个数为 2（小于 4）。
步骤 1: 点 [1 3] 被标记为噪声，因为其 eps 邻域内的点个数为 1（小于 4）。
聚类结果: [1, 2, 1, 1, 1, 2, 2, 2, -1, 1, 2, -1]
```

五、个人总结：

通过本次实验，我们学习了 DBSCAN（Density-Based Spatial Clustering of Applications with Noise）聚类算法的基本原理和实现方法。DBSCAN 是一种基于密度的聚类算法，能够识别任意形状的簇，并能够处理噪声数据。在实验中，我们首先理解了 DBSCAN 算法的核心概念，包括邻域半径 eps 和最小邻居数 min_samples 的作用。然后，我们编写了 Python 代码实现了 DBSCAN 算法，包括计算两点之间的欧氏距离、找到给定点的 eps 邻域内的所有点以及扩展簇等步骤。最后，我们使用示例数据集对 DBSCAN 算法进行了测试，并输出了聚类结果。

通过本次实验，我们加深了对 DBSCAN 算法的理解，掌握了如何使用 Python 实现该算法，并对基于密度的聚类方法有了更深入的认识。这些知识和经验将有助于我们在实际问题中应用聚类算法，并对数据进行更深入的分析和理解。