湖南科技大学计算机科学与工程学院

数据挖掘 课程设计报告

专业班级： 计算机科学与技术七班

姓　　名： 陈琪琪

学　　号： 2102010629

指导教师： 莫尚丰

时　　间： 2023.12.11-2023.12.15

地　　点： 逸夫楼 328

指导教师评语：	
成绩：　　等级：	签名： 年　月　日

目录

说明：“*”为自行添加内容。

实验一、Apriori 算法设计与应用

一、背景介绍

Apriori 算法是一种挖掘关联规则的频繁项集算法，其核心思想是通过候选集生成和向下封闭检测两个阶段来挖掘频繁项集。

二、实验内容

1. 实验要求

有如下的交易记录，请编写程序实现 Apriori 算法，挖掘该交易记录里的频繁项集，并挖掘出所有的强关联规则，设最小支持度为 50%，最小置信度为 50%。

交易记录	所购买的商品	交易记录	所购买的商品
1	A、B、C、D、E、F、G、	8	A、B、C、E、G、H、
2	A、B、C、D、E、H、	9	A、B、C、D、E、F、H、
3	A、B、C、D、E、F、G、H、	10	C、D、E、F、G、H、
4	A、B、C、G、H、	11	A、B、C、D、G、H、
5	A、B、C、D、G、H、	12	A、C、D、E、F、G、H、
6	A、B、C、D、E、F、G、H、	13	A、B、C、E、F、G、H、
7	A、B、C、D、E、F、G、	14	B、C、E、F、G、H、

2. 实验原理

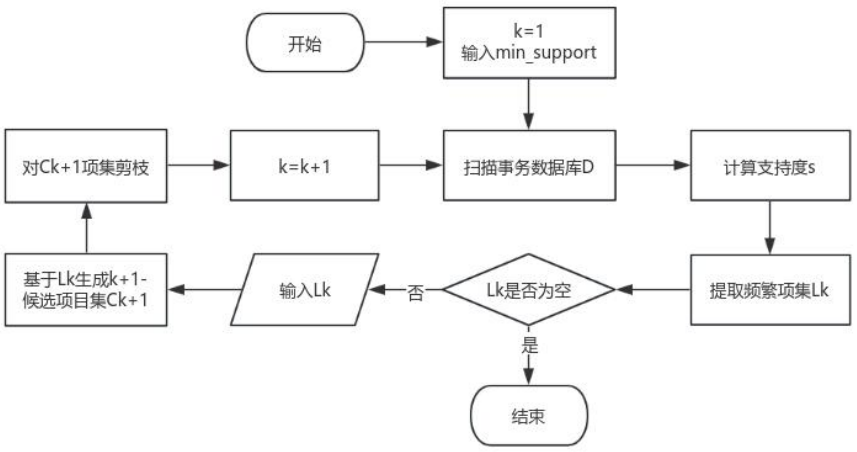
(1) 项目集空间理论

频繁项目集的非空子集是频繁项目集；非频繁项目集的超集是非频繁项目集。

(2) 基本原理

Apriori 算法通过迭代生成频繁项集，并基于频繁项集生成关联规则。算法中的关键步骤是通过支持度和置信度进行过滤，以保留满足阈值条件的项集和关联规则。由于 Apriori 算法使用了剪枝策略，可以在大规模数据集上有效地发现关联规则。

3. 程序流程图



4. 实验步骤

(1) **生成频繁项集：**Apriori 算法首先生成所有的 1-频繁项集，即每个项都是单独的元素。然后，通过迭代的方式生成更高阶的频繁项集，直到不能再生成为止。频繁项集是在数据集中出现频率达到设定阈值（支持度阈值）的项集。

(2) **生成关联规则：**对于每个频繁项集，从中生成关联规则。关联规则的形式为 A->B，表示在出现项集 A 的情况下，出现项集 B 的概率较大。关联规则的生成基于置信度的阈值，只有满足一定置信度的规则才会被保留。

(3) **支持度和置信度的计算：**支持度指的是一个项集在数据集中出现的频率，而置信度是关联规则的可信程度，即在 A 出现的情况下 B 出现的概率。在实验中，需要设定支持度和置信度的阈值，用于过滤具有一定频率和可信度的项集和关联规则。

(4) **剪枝操作：**在生成频繁项集的过程中，使用了 Apriori 原则来进行剪枝，即如果一个项集不是频繁的，那么它的所有超集也不是频繁的。这个原则帮助减少搜索空间，提高算法效率。

(5) **迭代：**上述步骤是迭代进行的，直到不能再生成新的频繁项集为止。

5. 关键源代码

(1) 基于 k-候选项目集C_k生成 k-频繁项目集L_k。

```
    def generate_Lk_by_Ck(dataSet, Ck, min_support, support_data):

        Lk = set()
        item_count = dict()
        for t in dataSet:
            for item in Ck:
                # 判断集合 item 的所有元素是否都包含在集合 t 中，即 item 是否是 t 的子集，即该候选项目集是否是遍历的数据的子集
                if item.issubset(t):
                    # 对选取的候选项的个数进行统计
                    # 对于 dict 中的 get 函数，如果能取到数据则返回字典中存储的 value，否则返回一个默认值，这里则是因为之前未记录所以返回 0 次
                    item_count[item] = item_count.get(item, 0) + 1

        t_num = float(len(dataSet))
        for item in item_count:
            # 计算支持度
            support = item_count[item] / t_num
            # 如果该支持度大于等于给定的最小支持度，则将其存放到 k-频繁项目集中去，并且保存到支持度数据字典中
            if support >= min_support:
                Lk.add(item)
                support_data[item] = support
        return Lk
```

(2) 生成强关联规则

```
    def generate_big_rules(L, support_data, min_conf):

        big_rule_list = []
        sub_set_list = []
```

```
        # 循环频繁项数据集 集合列表
        for i in range(0, len(L)):
            for freq_set in L[i]:
                for sub_set in sub_set_list:
                    if sub_set.issubset(freq_set):
                        # 置信度计算公式
                        conf = support_data[freq_set] / support_data[freq_set - sub_set]
                        big_rule = (freq_set - sub_set, sub_set, conf)
                        if conf >= min_conf and big_rule not in big_rule_list:
                            big_rule_list.append(big_rule)
                sub_set_list.append(freq_set)
        return big_rule_list
```

三、实验结果与分析

1. 实验结果

说明：因为生成的数据较多，这里仅展示 5-频繁项目集的强关联规则

强关联规则	置信度
['B', 'C', 'G', 'H']=>['A']	0.875
['A', 'B', 'C', 'G']=>['H']	0.778
['A', 'B', 'C', 'H']=>['G']	0.778
['A', 'B', 'G', 'H']=>['C']	1.000
['A', 'C', 'G', 'H']=>['B']	0.875
['A', 'C', 'G']=>['B', 'H']	0.700
['A', 'B', 'C']=>['G', 'H']	0.636
['B', 'C', 'H']=>['A', 'G']	0.700
['B', 'G', 'H']=>['A', 'C']	0.875
['C', 'G', 'H']=>['A', 'B']	0.700
['A', 'C', 'H']=>['B', 'G']	0.700
['B', 'C', 'G']=>['A', 'H']	0.700
['A', 'B', 'G']=>['C', 'H']	0.778
['A', 'G', 'H']=>['B', 'C']	0.875
['A', 'B', 'H']=>['C', 'G']	0.778
['C', 'H']=>['A', 'B', 'G']	0.583
['B', 'G']=>['A', 'C', 'H']	0.700
['A', 'G']=>['B', 'C', 'H']	0.700
['A', 'H']=>['B', 'C', 'G']	0.700
['A', 'C']=>['B', 'G', 'H']	0.583
['B', 'H']=>['A', 'C', 'G']	0.700
['G', 'H']=>['A', 'B', 'C']	0.700
['A', 'B']=>['C', 'G', 'H']	0.636
['B', 'C']=>['A', 'G', 'H']	0.583
['C', 'G']=>['A', 'B', 'H']	0.583
['H']=>['A', 'B', 'C', 'G']	0.583
['C']=>['A', 'B', 'G', 'H']	0.500

['G']=>['A', 'B', 'C', 'H']	0.583
['B']=>['A', 'C', 'G', 'H']	0.583
['A']=>['B', 'C', 'G', 'H']	0.583

2. 实验分析

(1) 由于实验给出的最小支持度为 50%以及最小置信度为 50%，但是由于数据集本身还是有很多条，所以生成的结果数据集较大，如果将最小支持度或最小置信度的阈值增大，其相应的频繁项集、候选项目集和强关联规则的数量也会减少。

(2) 算法根据项目集空间理论（频繁项目集的非空子集是频繁项目集；非频繁项目集的超集是非频繁项目集）从候选项目集上生成频繁项目集，以及根据最小支持度和最小置信度进行剪枝。

(3) Apriori 算法的两个致命的性能瓶颈:①多次扫描事务数据库，需要很大的 I/O 负载；②可能会产生庞大的候选集

四、小结与心得体会

在本次 Apriori 算法实验中，通过挖掘关联规则，我深刻体会到了其在发现数据关联性方面的强大能力。首先，通过设置适当的支持度和置信度阈值，我成功获得了一系列频繁项集和相关关联规则。这为我提供了对数据集模式和规律的深刻认识。其次，通过对挖掘结果的分析，我发现了一些有趣且实用的规则，这些规则在实际业务中具有潜在的应用价值。最后，我深感 Apriori 算法的可解释性和灵活性，尤其在商业智能和市场分析领域具有广泛应用前景。通过此次实验，我对数据挖掘的方法和应用有了更深入的理解，也为今后更复杂问题的解决提供了有力基础。

实验二、KNN 算法设计与应用

一、背景介绍

k-近邻（kNN, k-NearestNeighbor）是在训练集中选取离输入的数据点最近的 k 个邻居，根据这个 k 个邻居中出现次数最多的类别（最大表决规则），作为该数据点的类别。

二、实验内容

1. 实验要求

某班有 14 个同学，已登记身高及等级，新同学易昌，身高 1.74cm，等级是什么。请用 KNN 算法进行分类识别，其中 k=5。

序号	姓名	身高（cm）	等级
1	李丽	1.5	矮
2	吉米	1.92	高
3	马大华	1.7	中等
4	王晓华	1.73	中等
5	刘敏	1.6	矮
6	张强	1.75	中等
7	李秦	1.6	矮
8	王壮	1.9	高
9	刘冰	1.68	中等
10	张喆	1.78	中等
11	杨毅	1.70	中等
12	徐田	1.68	中等
13	高杰	1.65	矮
14	张晓	1.78	中等

2. 实验原理

(1) 距离度量

对于给定的测试样本，计算其与训练集中每个样本的距离。常用的距离度量包括欧氏距离、曼哈顿距离、闵可夫斯基距离等。

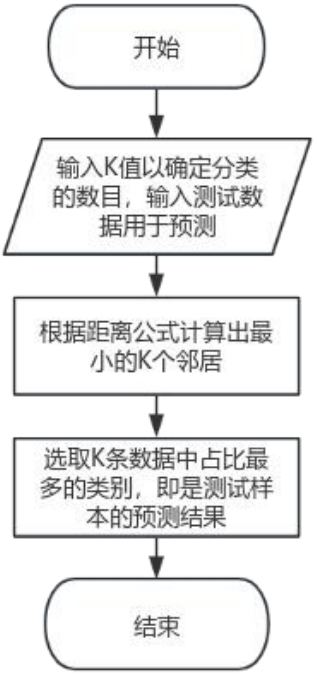
(2) 选择 K 个最近邻

从训练集中选择与测试样本距离最近的 K 个样本。

(3) 投票决策

对于分类任务，统计 K 个最近邻中各类别的出现次数，将测试样本划分为出现最多次的类别。对于回归任务，可以计算 K 个最近邻样本的平均值作为预测输出。

3. 程序流程图



4. 实验步骤

(1) **选择 K 值**：选择合适的 K 值，即最近邻居的数量。

(2) **距离度量**：选择适当的距离度量方法，如欧氏距离、曼哈顿距离等。这取决于数据的特性和问题的要求。

(3) **计算距离**：对于每个测试样本，计算其与训练集中每个样本的距离。使用选择的距离度量方法。

(4) **找到最近邻**：对计算得到的距离进行排序，并选择距离最近的 K 个训练样本作为最近邻。

(5) **投票决策**：对于分类问题，通过投票机制确定测试样本所属类别，即选择 K 个最近邻中出现最多次的类别。

5. 关键源代码

KNN 算法实现，入口参数分别是①需要分类的数据 input_data；②给定的数据集 dataSet；③数据集中分类的标签 labels；④分类的数目 k（实验选取的值为 5）。

```
    def KNN(input_data, dataSet, labels, k):

        # 计算身高差值，原理：通过调用函数让身高列表依次与给定身高进行相减，所得结果转为绝对值并保留两位小数
        list_height = np.round(list(map(abs, map(lambda x: x-input_data, dataSet[:, 1]))), 2)
        # np.argsort()根据元素的值进行降序排序并返回下标值
        sorted_index = np.argsort(list_height)

        # 定义一个字典用于存放各类型数量
        label_count = dict()
        # 只对前 k 个数据的类别数量进行统计
        for i in range(k):
```

```
            # 通过得出排序后的下标去映射其相应的类别
            level = labels[sorted_index[i]]
            # 对选取的K个数据所属的类别个数进行统计
            # 对于dict中的get函数，如果能取到数据则返回字典中存储的value，否则返回一个默认值，这里则是因为之前未记录所以返回0次
            label_count[level] = label_count.get(level, 0) + 1
        # 选取出现次数最多的类别
        max_count = 0
        result_level = ''
        for key, value in label_count.items():
            if value > max_count:
                max_count = value
                result_level = key

        return result_level
```

三、实验结果与分析

1. 实验结果

```
新同学【易昌】的身高等级为【中等】
```

2. 实验分析

(1) 由于实验给出 K 值为 5，从而选取的最近邻为 5 个。通过运行 KNN 算法，得到的最近邻中，标签为中等的数据有 5 条，即该五条数据均是中等身高，在标签数据中占比最大，从而【易昌】的身高等级为【中等】。

(2) 由于实验数据集中身高等级为中等的数据数量较多，从而在随着 K 值增大时，对于实验结果无明显影响。事实上，K 值的选取是与数据集本身有关。

(3) K 值的选取对结果的影响

①欠拟合与过拟合：当 k 值较小（如 k=1）时，模型更容易受到噪声和局部特征的影响，可能导致模型过拟合。反之，当 k 值较大时，模型更倾向于捕捉整体模式，可能导致欠拟合

②鲁棒性：较大的 k 值可以提高模型对噪声和异常值的鲁棒性，因为它不容易受到个别样本的影响。但在真实模式变化较快的情况下，过大的 k 值可能导致对局部特征的忽略

③计算复杂度：较大的 k 值意味着需要考虑更多的邻居，从而增加了计算复杂度。在大规模数据集上，选择合适的 k 值对于提高算法的效率至关重要。

四、小结与心得体会

在进行 KNN 算法实验的过程中，我深刻体会到其简单而强大的分类能力。不同数据集中通过调整 K 值和选择不同的距离度量都会对实验结果造成一定影响，由于本次实验数据量较少所以结果展示上无明显效果，但是更改数据集后这些模型参数的影响都得到了体现，因此不同问题需要选用适当的度量方法和 K 值。总体而言，KNN 算法在实验中展现了其直观、易理解的特点，同时也要在参数调优和对比实验中谨慎选择，以获得更好的性能。这次实验不仅加深了我对 KNN 算法的理解，也为我进一步探索模型优化和应用提供了指导。

实验三、ID3 算法设计与应用

一、背景介绍

信息增益是针对一个一个特征（属性）而言的，就是看一个特征，系统有它和没有它时的信息量各是多少，两者的差值就是这个特征给系统带来的信息量，即信息增益。ID3 算法：Iterative Dichotomiser 3，迭代二叉树 3 代，通过计算每个属性的信息增益，认为信息增益高的是好属性，每次划分选取信息增益最高的属性为划分标准，重复这个过程，直至生成一个能完美分类训练样例的决策树。

二、实验内容

1. 实验要求

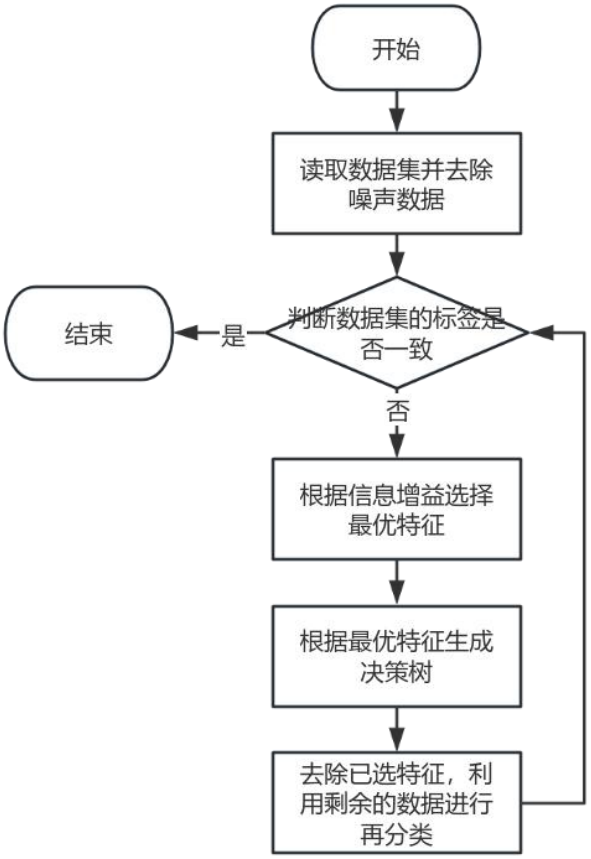
下表中共有 1024 条用户信息，分别表示不同年龄、收入、身份（是否为学生）、信誉的用户是否会购买电脑。请用 ID3 算法，构造相应的决策树。

计数	年龄	收入	学生	信誉	是否买电脑
64	青	高	否	良	不买
64	青	高	否	优	不买
128	中	高	否	良	买
60	老	中	否	良	买
64	老	低	是	良	买
64	老	低	是	优	不买
64	中	低	是	优	买
128	青	中	否	良	不买
64	青	低	是	良	买
132	老	中	是	良	买
64	青	中	是	优	买
32	中	中	否	优	买
32	中	高	是	良	买
63	老	中	否	优	不买
1	老	中	否	优	买

2. 实验原理

ID3 算法通过贪婪策略来选择信息增益最大的特征从而进行划分，逐步构建决策树。其实验原理基于对信息熵的优化，寻找最优的特征划分，以提高决策树在分类任务上的性能。

3. 程序流程图



4. 实验步骤

(1) **读取数据集并去除噪声数据：**对于一些给定条件相同但结果不一致的噪声数据要进行剔除以免影响实验结果。

(2) **特征选择：**选择一个最佳的特征作为当前节点的划分特征。ID3 算法使用信息增益来评估特征的好坏，信息增益高的特征被选为划分特征。

(3) **构建节点：**使用选定的特征创建一个节点，并将数据集划分为不同的子集，每个子集对应于该特征的一个取值。

(4) **递归：**对于每个子集，递归地应用 ID3 算法，直到达到停止条件，例如达到最大深度、节点包含的样本数量低于阈值等。

5. 关键源代码

(1) 去除噪声数据：对于一些给定条件都相同但结果不一致的数据，要进行处理，因为这些数据会对实验结果造成影响。本次实验通过判断给定条件是否一致，以及数据量相差是否悬殊从而判断其是否为噪声数据，是则进行剔除。

```
    # 先将数据中的噪声数据进行剔除，即给定的属性值相同但是结果不同且它们数量相差大的数据集删除，这里的比值设置为0.1
    # 用copy后的数据进行判断，在原数据中进行剔除
    for i in range(len(temp_copy)):
        for j in range(i+1, len(temp_copy)):
```

```
            if (np.array(temp_copy)[i, 1:-1] == np.array(temp_copy)[j, 1:
1]).all() and (float(np.array(temp_copy)[j, 0]) / float(np.array(temp_copy)
[i, 0])) <= 0.1:
                del(temp[j])
```

(2) 选择最优特征

```
    def chooseBestFeatureToSplit(dataSet, labels):
        num_data = float(len(dataSet))  # 数据集行数
        num_label = len(dataSet[0]) - 1  # 特征数量
        # 计数数据集的香农熵，即什么都没有做时根据已知数据集计算出来的熵
        shannonEnt = calcShannonEnt(dataSet)
        best_information_value = 0.0  # 将最佳信息增益初始化为0
        best_label_axis = -1  # 最优特征的索引值

        # 遍历所有特征，即遍历所有的列
        for i in range(num_label):
            # 获取 dataSet 的第 i 个所有特征，即 dataSet 的第 i 个属性的所有数据
            label_list = [example[i] for example in dataSet]
            # 创建 set 集合{}，元素不可重复，即将此作为分类标签
            label_set = set(label_list)
            condition_Ent = 0.0  # 经验条件熵，初始化条件熵为0
            # 计算信息增益
            for label in label_set:
                # 通过 set_after_split 获取划分后的子集，即获取从属于该分类标签
的数据集
                set_after_split = splitDataSet(dataSet, i, label)
                # 计算子集的概率
                prob = len(set_after_split)/num_data
                # 根据公式计算经验条件熵
                condition_Ent += prob*calcShannonEnt((set_after_split))
            # 计算信息增益的公式
            imformation_value = shannonEnt-condition_Ent
            # 打印出每个特征的信息增益
            print("分类属性%s 的信息增益
为%.3f" % (labels[i], imformation_value))
            if imformation_value > best_information_value:  # 比较出最佳信息
增益
                # 更新信息增益，找到最大的信息增益
                best_information_value = imformation_value
                # 记录信息增益最大的特征的索引值
                best_label_axis = i

        # 返回信息增益最大特征的索引值
        return best_label_axis  # 返回已使用的分类属性
```

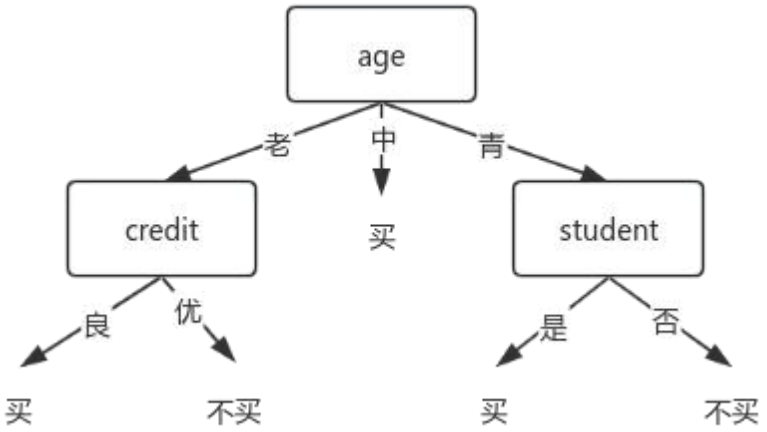
(3) 实验改进：C4.5 算法。C4.5 算法则是在 ID3 算法的基础上将最优特征的衡量标准信息增益更改为信息增益比例，由于 splitI 值可能会取 0 值从而导致程序抛异常且无判定价值，因此需要额外判定进而优化算法。

```
    if splitInfo == 0:
        print("分类属性%s 的 splitI 值为 0，从而该属性无参考价值需排除
" % (labels[i]))
        continue
        # 计算第 i 列属性的信息熵益比例
        gainRatio = (shannonEnt-condition_Ent)/splitInfo
        # 打印出每个特征的信息增益
        print("分类属性%s 的信息增益比例为%.3f" % (labels[i], gainRatio))
        if gainRatio > best_gainRatio_value:  # 比较出最佳信息增益比例
            # 更新信息增益比例，找到最大的信息增益
            best_gainRatio_value = gainRatio
            # 记录信息增益比例最大的特征的索引值
            best_label_axis = i
```

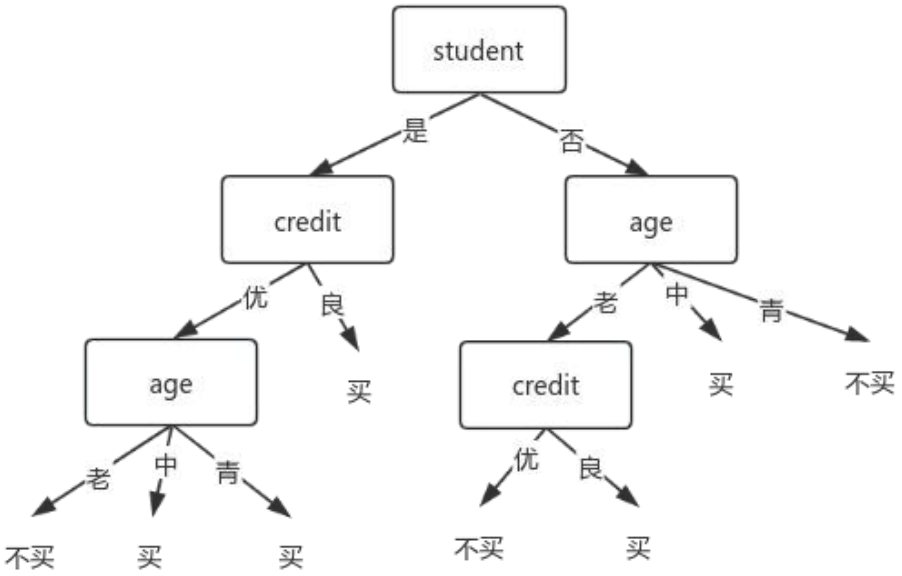
三、实验结果与分析

1. 实验结果

ID3 算法实验结果：



C4.5 算法实验结果：



2. 实验分析

(1) ID3 算法的优缺点

- **算法优点：**
  - ✧ 算法的理论清晰，方法简单，学习能力较强。
  - ✧ ID3 算法的假设空间包含所有的决策树，它是关于现有属性的有限离散值函数的一个完整空间。所以 ID3 算法避免了搜索不完整假设空间的一个主要风险：假设空间可能不包含目标函数。
  - ✧ ID3 算法在搜索的每一步都使用当前的所有训练样例，大大降低了对个别训练样例错误的敏感性。因此，通过修改终止准则，可以容易地扩展到处理含有噪声的训练数据。
- **算法缺点：**
  - ✧ 只对比较小的数据集有效，且对噪声比较敏感，当训练数据集加大时，决策树可能会随之改变。
  - ✧ ID3 算法在搜索过程中不进行回溯。收敛到局部最优而不是全局最优。
  - ✧ ID3 算法只能处理离散值的属性。信息增益度量存在一个内在偏置，它偏袒具有较多值的属性。如日期属性。

(2) 与 C4.5 算法的异同

- **相同点**：C4.5 算法同 ID3 算法类似，只不过在计算过程中采用信息增益率来选择特征而不是信息增益。通过引入信息增益比，一定程度上对取值比较多的特征进行惩罚，避免出现过拟合的特性，提升决策树的泛化能力。
- **不同点**：C4.5 算法相对 ID3 算法增加了新功能，如①用信息增益比例的概念；②合并具有连续属性的值；③可以处理缺少属性值的训练样本；④通过使用不同的修剪技术以避免树的过度拟合；⑤k 交叉验证；⑥规则的产生方式。

四、小结与心得体会

在进行 ID3 算法实验的过程中，我深刻认识到其在构建决策树方面的优越性。通过选择最佳特征进行信息增益计算，ID3 算法能够有效地构建具有较好泛化能力的分类模型。实验过程中我也意识到在处理大规模和高维度数据时，ID3 算法可能面临计算复杂度的挑战，这需要在实际应用中加以考虑。除了实现 ID3 算法之外，我又实现了 C4.5 算法并对其进行优化，通过对 C4.5 算法的实现，让我对 C4.5 算法与 ID3 算法之间的异同点得到了进一步的理解。总体而言，ID3 算法为我提供了一种强大的工具，帮助我解决分类问题，并对决策树算法的原理有了更深层次的理解。

实验四、DBSCAN 算法设计与应用

一、背景介绍

DBSCAN 算法：如果一个点 q 的区域内包含多于 MinPts 个对象，则创建一个 q 作为核心对象的簇。然后，反复地寻找从这些核心对象直接密度可达的对象，把一些密度可达簇进行合并。当没有新的点可以被添加到任何簇时，该过程结束。

二、实验内容

1. 实验要求

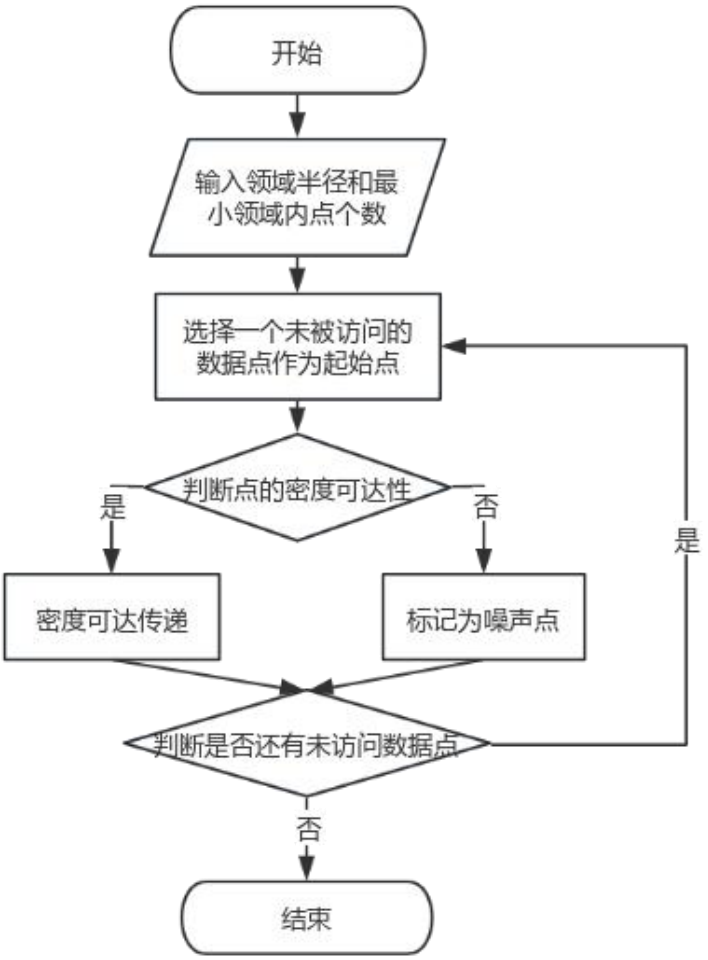
有如下二维数据集，取 ε = 2，minpts = 3，请使用 DBSCAN 算法对其聚类（使用曼哈顿距离）

序号	横坐标	纵坐标	序号	横坐标	纵坐标	序号	横坐标	纵坐标
1	20	45	22	20	45	43	6	36
2	34	23	23	34	23	44	78	37
3	53	67	24	53	67	45	7	38
4	54	85	25	54	85	46	70	39
5	67	4	26	67	4	47	6	36
6	33	67	27	33	67	48	45	45
7	24	78	28	24	78	49	67	67
8	37	90	29	37	90	50	84	6
9	67	34	30	67	34	51	23	78
10	34	56	31	78	32	52	13	7
11	89	78	32	23	33	53	45	70
12	65	23	33	45	34	54	67	76
13	45	45	34	67	35	55	4	54
14	67	67	35	67	76	56	68	60
15	84	6	36	4	54	57	45	45
16	23	78	37	68	60	58	34	67
17	13	7	38	7	38	59	35	67
18	45	70	39	70	39	60	36	4
19	67	76	40	76	40	61	37	68
20	4	54	41	45	70	62	38	45
21	68	60	42	67	76	63	20	34

2. 实验原理

DBSCAN 算法实验首先设定邻域半径 ε 和最小邻域内点数 MinPts 参数，然后从未被访问的数据点开始，根据密度可达性原则，将点标记为核心点并递归地扩展其密度可达的点，形成聚类簇。噪声点通过未满足 MinPts 条件的点识别。通过调整参数，实验能发现不同形状和密度的聚类，具有对噪声点的鲁棒性。最终，DBSCAN 输出聚类结果及噪声点，对于发现不规则形状的聚类结构具有优越性。

3. 程序流程图



4. 实验步骤

(1) **参数设置**：定义算法的核心参数，包括邻域半径（ε）和最小邻域内点个数（MinPts）。

(2) **初始化**：选择一个未被访问的数据点作为起始点。

(3) **密度可达性判定**：计算起始点的邻域内点数，若大于等于 MinPts，则将其标记为核心点。

(4) **密度可达传递**：对核心点，递归地扩展其密度可达的点，将它们加入同一聚类簇。即，找到邻域内的点，将其也标记为核心点，然后递归地找到它们的邻域内的点，直到无法再找到新的核心点。

(5) **噪声点标记**：对于未被标记的数据点，如果其邻域内的点数小于 MinPts，则将其标记为噪声点。

(6) **迭代扩展**：重复以上过程，选择下个未被访问的数据点直到所有数据点都被访问。

(7) **聚类形成**：将所有标记为核心点的点及其密度可达的点归为一个聚类。未被归为任何聚类的点被视为噪声点。

(8) **输出结果**：输出最终的聚类结果，包含若干个聚类簇以及噪声点。

5. 关键源代码

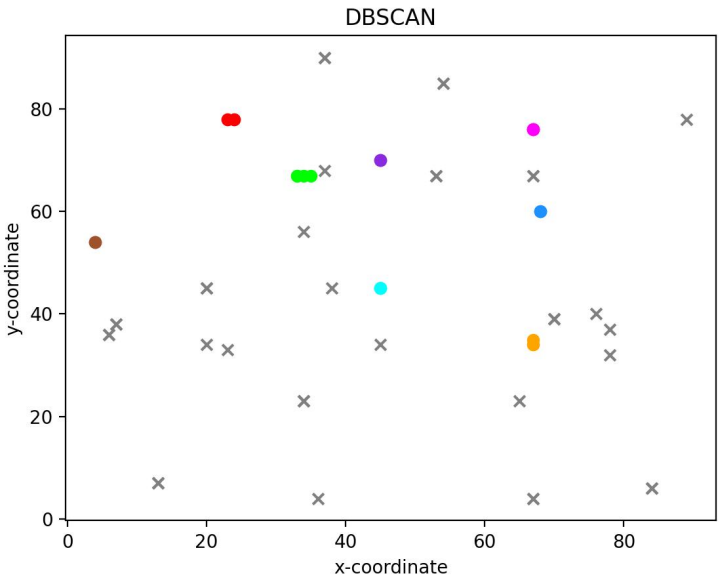
DBSCAN 算法，入口参数分别是①数据集；②Eps 为指定半径参数；③MinPts 为指定邻域密度阈值。

```
    def dbscan(Data, Eps, MinPts):
        num = len(Data)  # 点的个数
        unvisited = [i for i in range(num)]  # 没有访问到的点的列表
        visited = []  # 已经访问的点的列表
        C = [-1 for i in range(num)]
        # C 为输出结果，默认是一个长度为 num 的值全为-1 的列表
        # 用 k 来标记不同的簇，k = -1 表示噪声点
        k = -1
        # 如果还有没访问的点
        while len(unvisited) > 0:
            # 随机选择一个 unvisited 对象
            p = random.choice(unvisited)
            unvisited.remove(p)
            visited.append(p)
            # N 为 p 的 epsilon 邻域中的对象的集合
            N = []
            for i in range(num):
                if (dist(Data[i], Data[p]) <= Eps):  # and (i!=p):
                    N.append(i)
            # 如果 p 的 epsilon 邻域中的对象数大于指定阈值，说明 p 是一个核心对象
            if len(N) >= MinPts:
                k = k + 1
                C[p] = k
                # 对于 p 的 epsilon 邻域中的每个对象 pi
                for pi in N:
                    if pi in unvisited:
                        unvisited.remove(pi)
                        visited.append(pi)
                        # 找到 pi 的邻域中的核心对象，将这些对象放入 N 中
                        # M 是位于 pi 的邻域中的点的列表
                        M = []
                        for j in range(num):
                            if (dist(Data[j], Data[pi]) <= Eps):  # and (j!=pi):
                                M.append(j)
                        if len(M) >= MinPts:
                            for t in M:
                                if t not in N:
                                    N.append(t)
                    # 若 pi 不属于任何簇，C[pi] == -1 说明 C 中第 pi 个值没有改动
                    if C[pi] == -1:
                        C[pi] = k
            # 如果 p 的 epsilon 邻域中的对象数小于指定阈值，说明 p 是一个噪声点
            else:
                C[p] = -1
        return C
```

三、实验结果与分析

1. 实验结果

簇	簇所含点的序号
Cluster[1]	19，35，42，54
Cluster[2]	6，27，58，59
Cluster[3]	21，37，56
Cluster[4]	9，30，34
Cluster[5]	13，48，57
Cluster[6]	18，41，53
Cluster[7]	20，36，55
Cluster[8]	7，16，28，51
Noise	1，2，3，4，5，8，10，11，12，14，15，17，22，23，24，25，26，29，31，32，33，38，39，40，43，44，45，46，47，49，50，52，60，61，62，63



2. 实验分析

(1) 数据集选择对实验的影响：选择不同类型的数据集，包括具有不同形状、不同密度分布和不同噪声水平的数据有助于了解 DBSCAN 对于不同类型数据的适应性。

(2) 参数调优：对 DBSCAN 的参数进行调优，主要是邻域半径（ ε ）和最小样本数（MinPts）。通过实验确定最佳参数设置，可以获得最好的聚类效果。

(3) 噪声处理能力：DBSCAN 具有较强的噪声处理能力，主要因为它将噪声点识别为不属于任何簇的孤立点。

四、小结与心得体会

在进行 DBSCAN 算法实验后，我深刻体会到其在聚类任务上的优越性。DBSCAN 通过定义样本点周围的密度来发现聚类结构，克服了 K 均值等算法对聚类形状和密度敏感的限制。实验中，DBSCAN 能够自动识别出不同形状和大小的簇，对噪声点表现出鲁棒性。相较于传统的聚类算法，DBSCAN 的参数设置相对较少，对于数据集的密度变化具有较好的适应性。然而，我也发现在一些特定场景下，如数据集中簇的密度差异很大或者包含大量噪声时，DBSCAN 的表现可能受到影响，需要进行参数调整以获得更好的效果。总的来说，DBSCAN 算法在聚类任务中展现了强大的能力，尤其适用于挖掘具有不规则形状和变化密度的簇。通过实验，我更深刻地理解了 DBSCAN 的工作原理和适用范围，这为我更好地选择合适的聚类算法提供了指导。

实验五、K-means 算法设计与应用

一、背景介绍

K-means 算法，也被称为 K-平均或 K-均值，是一种得到最广泛使用的聚类算法。相似度的计算根据一个簇中对象的平均值来进行。算法首先随机地选择 K 个对象，每个对象初始地代表了一个簇的平均值或中心。对剩余的每个对象根据其与各个簇中心的距离，将它赋给最近的簇。然后重新计算每个簇的平均值。这个过程不断重复，直到准则函数收敛。

二、实验内容

1. 实验要求

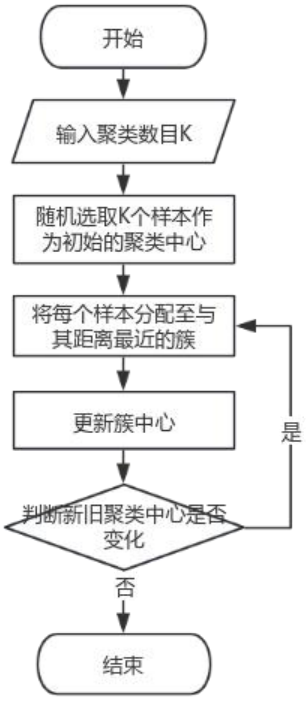
请对下表中的数据进行 K-means 聚类，距离为欧氏距离，k=3.

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1	2	2	4	5	6	6	7	9	1	3	5	3
2	1	4	3	8	7	9	9	5	12	12	12	3

2. 实验原理

K-means 聚类实验首先准备数据集，确定聚类数 K。随机或智能初始化 K 个簇中心，迭代地分配样本到最近的簇，然后更新簇中心。通过迭代优化，判断算法是否收敛。最终，对聚类结果进行评估和可视化。实验包括选择合适的 K 值、初始化策略、收敛条件，并通过内外部指标评价聚类效果。通过调整参数，可以深入理解 K 均值算法的性能和适用性。

3. 程序流程图



4. 实验步骤

(1) **确定聚类数目 K：**在进行 K 均值聚类之前，需要确定簇的数量 K。

(2) **初始化聚类中心：**随机选择 K 个样本作为初始的聚类中心，或者使用其他初始化方法来提高初始聚类中心的质量。

(3) **迭代优化：**通过以下步骤迭代地优化聚类结果。

a) 分配样本到簇：计算每个样本到各个聚类中心的距离，将样本分配到距离最近的簇。

b) 更新簇中心：对每个簇，计算其所有样本的平均值，并将该平均值作为新的聚类中心。

(4) **收敛检验：**判断算法是否收敛，即判断新的聚类中心和旧的聚类中心之间的变化是否小于某个阈值。如果满足停止条件，则算法结束，否则返回第 3 步。

5. 关键源代码

k-means 算法，入口参数分别是①数据集；②聚类数目 K；③簇类中心。

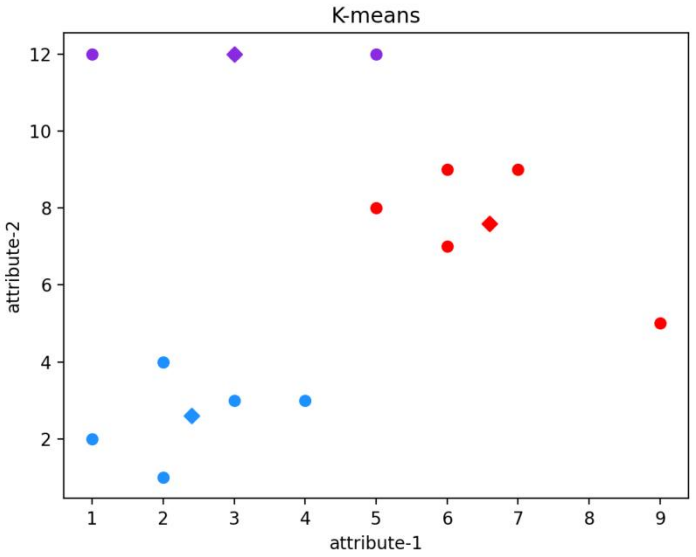
```
    def kmeans(dataSet, k, centroids):
        groups = [[] for _ in range(k)]    # 创建存放 k 组数据的二维列表
        print(centroids)  # 打印每次迭代的质心列表
        # 将各数据进行分组
        for data in dataSet:
            diff = np.tile(data, (k, 1)) - centroids    # 计算各点与参考点的距离
            absDist = np.sum(np.abs(diff), axis=1)  # 计算与各质点绝对值之和(axis=1 表示行)的距离
            group_index = list(absDist).index(np.min(absDist))  # 所属于哪组的索引
            groups[group_index].append(data)  # 将该点添加到该组中去

        newCentroids = []    # 初始化新质心列表
        # 计算各组质心
        for group in groups:
            # 计算分组后的新的质心
            newCentroids.append(list(np.mean(group, axis=0)))  # axis=0 表示列

        flag = (newCentroids == centroids)  # 判断新旧质心组是否改变

        if flag == False:  # 如果改变了递归 kmeans 函数，否则就返回该质心和分组
            return kmeans(dataSet, k, newCentroids)
        return newCentroids, groups
```

三、实验结果与分析

1. 实验结果

说明：本次实验初始化聚类中心环节随机取 k 个点作为初始点。以下实验结果展示为出现最频繁的结果；图形展示中，菱形点表示为簇的质心。

簇	质心
[1, 12], [3, 12], [5, 12]	[3.0, 12.0]
[1, 2], [2, 1], [2, 4], [4, 3], [3, 3]	[2.4, 2.6]
[5, 8], [6, 7], [6, 9], [7, 9], [9, 5]	[6.6, 7.6]



2. 实验分析

(1) K-means 算法的优缺点

- **算法优点**
  - ✧ 是解决聚类问题的一种经典算法，简单、快速。
  - ✧ 对处理大数据集，该算法是相对可伸缩和高效率的。
  - ✧ 当结果簇是密集的，它的效果较好。
- **算法缺点**
  - ✧ 在簇的平均值被定义的情况下才能使用，可能不适用于某些应用。
  - ✧ 必须事先给出 k（要生成的簇的数目），而且对初值敏感，对于不同的初始值，可能会导致不同结果。
  - ✧ 不适合于发现非凸面形状的簇或者大小差别很大的簇。而且，它对于“躁声”和孤立点数据是敏感的。

(2) K-means 算法的时间复杂度和空间复杂度分析

- **时间复杂度**：K-means 算法的时间消耗体现在初始化、迭代优化和收敛检验，但在通常情况下，K-means 算法的总体时间复杂度主要由迭代优化阶段的计算决定，即 O(iter * n * K * d)，其中 iter 是迭代次数。
- **空间复杂度**：K-means 算法的空间消耗体现在存储数据、存储簇中心和距离矩阵。它的总体空间复杂度为 O(n * d + K * d + n^2 + n)，其中前两项主要占用存储数据和簇中心的空间，第三项是距离矩阵的空间，第四项是簇分配信息的空间。

四、小结与心得体会

在 K-means 算法的实验中，我深刻认识到其对于聚类任务的强大性能。通过调整聚类数 K、距离公式以及迭代次数，我发现这些参数对聚类结果影响深远。实验中，K-means 算法在处理规模较大的数据集时表现出较高的效率，但对于异常值敏感。同时，K-means 算法对于凸形簇的表现良好，但在处理非凸形簇或密度不均匀的数据时可能存在局限。总体而言，实验让我更深入了解 K-means 算法的优势和限制，为合理选择聚类算法提供了重要经验。

实验六、PageRank 算法设计与应用

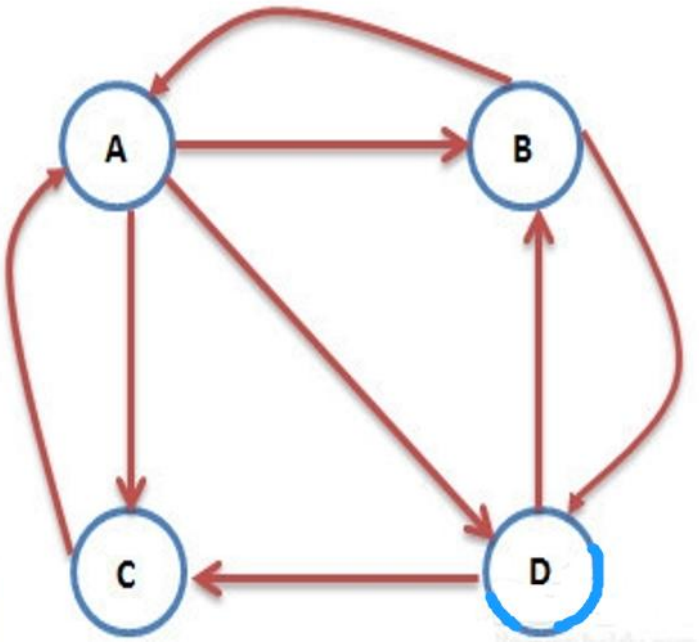
一、背景介绍

PageRank 算法：计算每一个网页的 PageRank 值，然后根据这个值的大小对网页的重要性进行排序。

二、实验内容

1. 实验要求

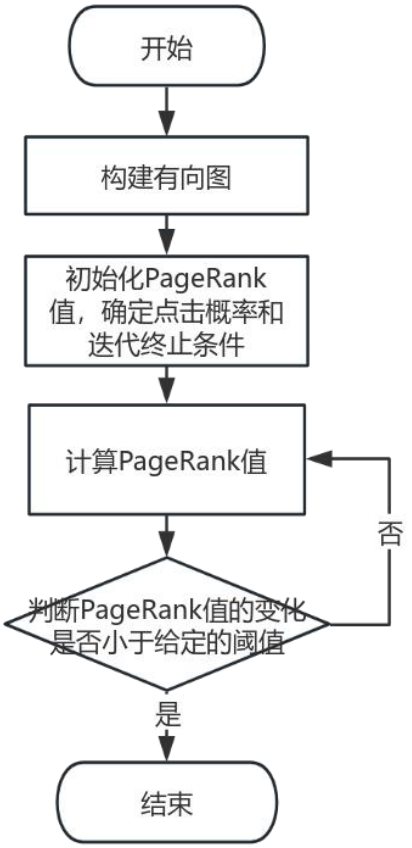
互联网中的网页的链接可以看作是有向图，如 A、B、C、D 四个网页，请采用 PageRank 算法对网页进行分级排序。



2. 实验原理

PageRank 算法实验首先构建图模型，其中节点表示页面，边表示页面间的超链接关系。初始化每个页面的 PageRank 值，然后通过迭代计算更新每个页面的 PageRank 值。更新规则考虑页面的入链和出链，以及它们的 PageRank 值。实验中需要确定收敛条件，通常是设定最大迭代次数或设定 PageRank 值的变化小于某个阈值。实验还可能涉及调整阻尼因子的影响，该因子控制跳转到其他页面的概率。

3. 程序流程图



4. 实验步骤

(1) **构建图模型：**从实际应用中获取网页数据，构建有向图，其中节点表示网页，边表示网页之间的超链接关系。

(2) **初始化 PageRank 值：**为每个网页初始化 PageRank 值，可以采用均匀分布或其他策略。初始值可以是相等的，也可以基于先验知识设置。

(3) **迭代计算 PageRank 值：**通过迭代更新每个网页的 PageRank 值。更新规则考虑页面的入链和出链，以及它们的 PageRank 值。PageRank 的更新公式通常涉及阻尼因子，控制跳转到其他页面的概率。

(4) **设定收敛条件：**设定迭代停止的条件，可以是设定最大迭代次数或设定 PageRank 值的变化小于某个阈值。

5. 关键源代码

PageRank 算法，入口参数分别是①邻接矩阵 A；②阻尼因子 d。

在读取的数据集中，读取的是页面的有向图，即数据集中存放的是有向图的表示，只不过由原先的正向链转为了反向链。

```
    def pageRank(A, d):  # d 为阻尼因子
        # shape[0]是行数，shape[1]是列数
        R0 = np.ones(A.shape[1])  # R0 为初始等级值向量
        Q = np.ones((A.shape[0], A.shape[1])) / A.shape[0]  # Q 为常量矩阵
        M = np.dot((1 - d), Q) + np.dot(d, A)  # M 为转移概率矩阵

        iter_term_condition = 0.1  # 迭代终止条件
```

```
        iter_condition = float('inf')  # 两向量逐分量和，初始化为无穷大
        R2 = R0
        # 直到值小于等于迭代终止条件时结束迭代
        while (iter_condition > iter_term_condition):
            R1 = R2
            R2 = np.dot(M, R1)
            iter_condition = np.sum(np.fabs(R2 - R1))
        R2 = np.round(R2, decimals=2)  # 保存两位小数

        return R2
```

三、实验结果与分析

1. 实验结果

说明：由于实验要求中没有给出阻尼因子 d 的值以及迭代的终止条件，因此本次实验采取的是 d=0.85，迭代终止条件是 0.1。

```
A的等级值:1.29
B的等级值:0.9
C的等级值:0.9
D的等级值:0.9
```

2. 实验分析

(1) PageRank 算法的优缺点

- **算法优点**
  - ✧ 链接结构分析：PageRank 通过分析页面之间的链接结构，能够较好地反映网页的重要性和影响力。
  - ✧ 全局性质：PageRank 考虑整个图结构，具有全局性质，适用于评估整个网络中节点的重要性。
  - ✧ 应用广泛：PageRank 最初用于搜索引擎中网页的排名，但它的思想也在其他领域广泛应用，如社交网络分析、推荐系统等。
- **算法缺点**
  - ✧ 主观性：PageRank 假设用户是按照某种概率浏览页面，这种模型可能不准确，特别是在用户行为多样性较大的情况下。
  - ✧ 静态模型：原始 PageRank 是一个静态模型，不考虑时间动态性，无法很好地处理实时变化的网络。
  - ✧ 不考虑内容：PageRank 只基于链接结构，不考虑网页的内容信息，对一些特殊场景的适应性有限。

(2) PageRank 算法的时间复杂度和空间复杂度分析

- **时间复杂度**：PageRank 算法的时间复杂度取决于初始化阶段、迭代计算阶段和迭代次数，所以 PageRank 算法的总体计算复杂度为 O(iter * (N + E))，其中 iter 是迭

代次数。

- **空间复杂度**：PageRank 算法的空间复杂度主要由存储每个网页的 PageRank 值和图的链接结构所需的空间决定，为 O(N + E)。

注意：PageRank 算法的计算复杂度随着图的规模增大而增加，对于大规模的网络结构，可能需要采用分布式计算等技术来提高计算效率。此外，一些优化技术，如稀疏矩阵计算和并行化计算，也可以用来降低 PageRank 算法的计算开销。

四、小结与心得体会

在 PageRank 算法的实验中，我深入研究了其在网络图中评估节点重要性的效果。通过构建有向图、初始化节点 PageRank 值，逐步迭代计算节点的 PageRank，实验过程中需要精心选择迭代终止条件和阻尼因子等参数。通过实验，我发现 PageRank 对于反映网络中节点的相对重要性和影响力具有很好的效果，尤其适用于大规模网络。然而，实验也揭示了 PageRank 算法的一些局限性，特别是对于动态网络和内容相关性的考虑不足。在调优实验中，我通过改变迭代次数、调整阻尼因子等参数，深刻理解了这些参数对 PageRank 结果的影响。总体而言，PageRank 算法实验使我更深刻地理解了其原理、应用范围和局限性，为在实际应用中灵活运用 PageRank 提供了宝贵经验。

实验七、模型评估指标

一、背景介绍

当模型训练完成后，需要进行模型评估来量化模型性能的好坏。而且模型的性能不单单只有一个维度，所以模型的好坏通常会用多个指标来进行衡量。例如，现在想要衡量一个分类器的性能，可能第一时间会想到用准确率来衡量模型的好坏，但是准确率高并不一定就代表模型的性能高，因此可能会需要使用如 F1-Score、AUC 等指标来衡量。

二、实验内容

1. 实验要求

请编写程序，实现混淆矩阵、准确率、精确率、ROC 曲线、AUC 面积、F1-Score 的计算，对下列数据进行分析。

(1) 某分类器对 66 只动物进行分类，其中 13 只猫，53 只不是猫，分类器判断时这 13 只猫只有 10 只预测对了，其他动物也只预测对了 45 只。

(2) 某分类器对图片进行分类，共有 100 张图片。其中 36 张汽车，预测对了 30 张，另外 4 张预测成了船，2 张预测成了房子。40 张船，预测对了 34 张，另外 6 张预测成了汽车。24 张房子，预测对了 24 张。

2. 实验原理

(1) 混淆矩阵

对于二分类问题，可将样例根据其真实类别与分类器预测类别的组合划分为：

真正例(True Positive，TP)，将正类预测为正类的数量，即预测正确；

真负例(True Negative，TN)，将负类预测为负类的数量，即预测正确；

假正例(False Positive，FP)，将负类预测为正类的数量，即预测错误--误报；

假负类(False Negative，FN)，将正类预测为负类的数量，即预测错误--漏报。

令 TP、TN、FP、FN 分别表示其对应的样本数，则 TP + TN + FP + FN = 样本总数，分类的"混淆矩阵"如下：

真实情况	预测情况	
	正类	反类
正类	TP（真正例）	FN（假负例）
反类	FP（假正例）	TN（真负例）

(2) 准确率(Accuracy)：预测正确的样本数(TP 和 TN)占所有样本数的比例，一般地说，准确率越高，分类器越好，计算公式如下：

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

(3) 精确率(Precision)，亦称查准率：正确预测为正的样本数(TP)占全部预测为正(TP 和 FP)的比例，计算公式如下：

$$Precision = \frac{TP}{TP + FP}$$

(4) 召回率(Recall)，亦称查全率：正确预测为正的样本数(TP)占全部实际为正(TP 和 FN)的比例，计算公式如下：

$$Recall = \frac{TP}{TP + FN}$$

总结：精确率和召回率是一对矛盾的度量，一般来说，精确率高时，召回率往往偏低；而召回率高时，精确率往往偏低。

(5) F1-Score：F1 值为算数平均数除以几何平均数，且越大越好，计算公式如下：

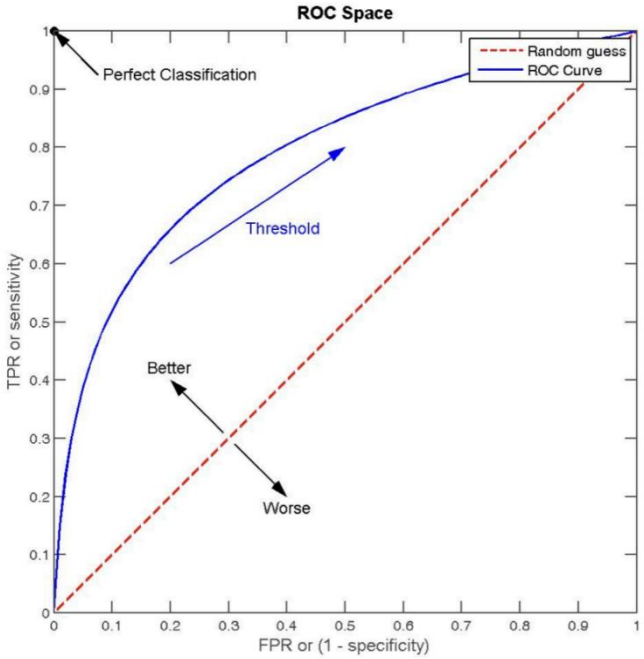
$$F_1 - Score = \frac{2 \times Precision \times Recall}{Precision + Recall} = \frac{2TP}{2TP + FP + FN}$$

(6) ROC 曲线，全称是“受试者工作特征”曲线，是研究学习器泛化性能的有力工具。ROC 曲线的纵轴是“真正例率”（True Positive Rate，简称 TPR），预测为正实际为正的样本数（TP）占全部实际为正（TP 和 FN）的比例；横轴是“假正例率”（False Positive Rate，简称 FPR），预测为正实际为负的样本数（FP）占全部实际为负（TN 和 FP）的比例。计算公式如下：

$$TPR = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{TN + FP}$$

如图所示：



横轴 FPR：FPR 越大，预测正类中实际负类越多。

纵轴 TPR：TPR 越大，预测正类中实际正类越多。

理想目标：TPR=1，FPR=0,即图中(0,1)点，故 ROC 曲线越靠拢(0,1)点，越偏离 45 度对角线越好，TPR、1-FPR 越大效果越好。

(7) AUC 值（Area Under ROC Curve），是指 ROC 曲线下的面积。从数学角度讲，AUC 值是小于 1 的数值，一般情况下，ROC 曲线都位于 y=x 这条直线的上方，所以 AUC 是介于 0.5-1 之间。

从 AUC 判断分类器（预测模型）优劣的标准：

✧ AUC = 1，是完美分类器，采用这个预测模型时，存在至少一个阈值能得出完美预测。绝大多数预测的场合，不存在完美分类器。

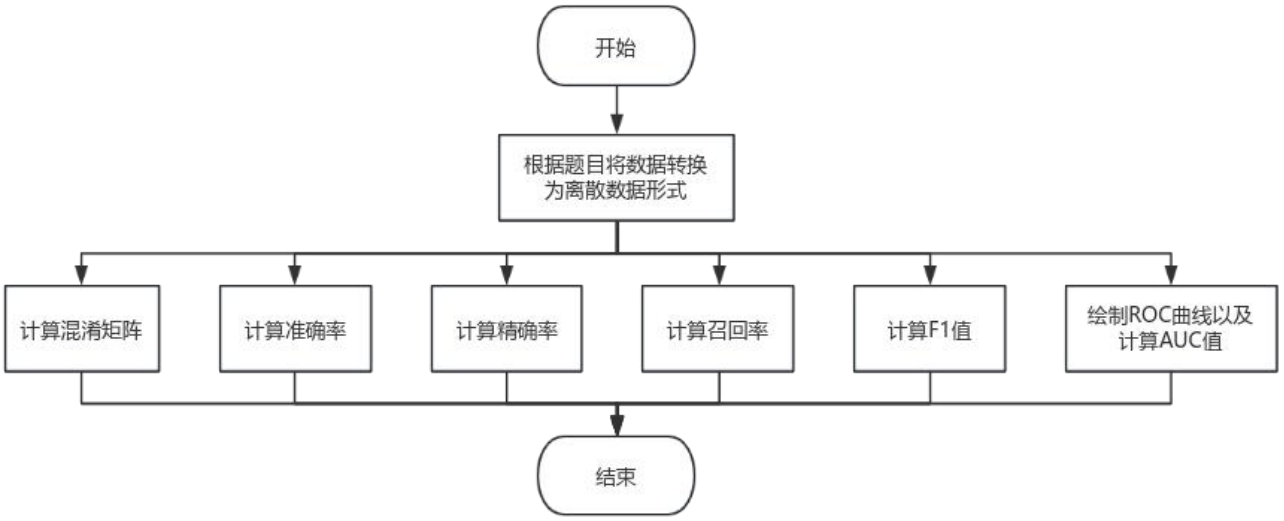
✧ 0.5 < AUC < 1，优于随机猜测。这个分类器（模型）妥善设定阈值的话，能有预测价值。

✧ AUC = 0.5，跟随机猜测一样（例：抛硬币），模型没有预测价值。

✧ AUC < 0.5，比随机猜测还差；但只要总是反预测而行，就优于随机猜测。

总之，AUC 值越大的分类器，正确率越高。

3. 程序流程图



4. 实验步骤

(1) **转换数据形式：**根据题目将数据转换为离散数据形式。

(2) **计算相关评估指标：**计算混淆矩阵、准确率、精确率、召回率、$F_1 - Score$、绘制 ROC 曲线以及计算 AUC 值。

5. 关键源代码

模型的评估指标精确率、召回率、准确率、F1-score、ROC 曲线和 AUC 值。

```
1.      # 定义精确率
2.      def precision(tp, fp, fn, tn):
3.          return round(float(tp) / (tp + fp), 3)
4.
5.      # 定义召回率
6.      def recall(tp, fp, fn, tn):
7.          return round(float(tp) / (tp + fn), 3)
8.
9.      # 定义准确率
10.     def accuracy(tp, fp, fn, tn):
11.         return round(float(tp + tn) / (tp + fp + fn + tn), 3)
12.
13.     # 定义f1_score
14.     def f1_score(tp, fp, fn, tn):
```

```
        return round(float(2 * precision(tp, fp, fn, tn) * recall(tp, fp, fn, tn)) / (precision(tp, fp, fn, tn) + recall(tp, fp, fn, tn)), 3)

    # 计算roc和auc
    def calculate_roc_auc(data_true, data_predict):
        fpr, tpr, threshold = roc_curve(data_true, data_predict)
        result_auc = round(auc(fpr, tpr), 3)
        print(f'auc:{result_auc}')

        lw = 2
        plt.subplot(1, 1, 1)
        # 假正率为横坐标，真正率为纵坐标做曲线
        plt.plot(fpr, tpr, color='red', lw=lw, label='roc_curve(auc=%0.3f)' % result_auc)
        plt.plot([0, 1], [0, 1], color='blue', lw=lw, linestyle='--')
        # 限制x，y轴的范围为[0,1]
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.0])
        # 将x轴命名为fpr，y轴命名为tpr
        plt.xlabel('fpr')
        plt.ylabel('tpr')
        plt.title('ROC', y=0.5)
        plt.legend(loc="lower right")
        plt.show()
```

三、实验结果与分析

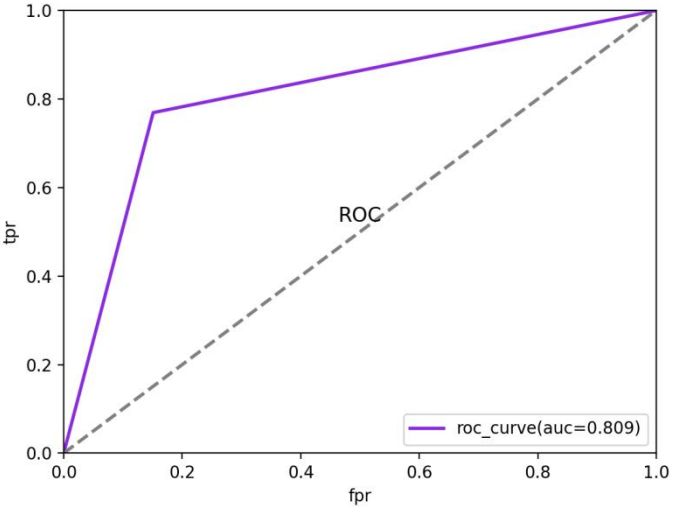
1. 实验结果

数据集 1 实验结果

其他评估指标结果：

Confusion Matrix	TP:10	FN:8
	FP:3	TN:45
Precision	0.769	
Accuracy	0.833	
Recall	0.556	
F1_score	0.645	
AUC	0.809	

ROC 曲线：



数据集 2 实验结果

Sklearn 库中调用函数 precision, recall, f1-score 的属性 average 取值说明：

(1)micro：将所有类别的 True Positive、False Positive 和 False Negative 汇总，然后计算精确率、召回率和 F1-Score。适用于类别不平衡的情况。

(2)macro：对每个类别分别计算精确率、召回率和 F1-Score，然后取平均值。适用于每个类别的权重相同的情况。

(3)weighted：对每个类别分别计算精确率、召回率和 F1-Score，然后按照每个类别的样本数进行加权平均。适用于类别不平衡的情况，且更关注样本数多的类别。

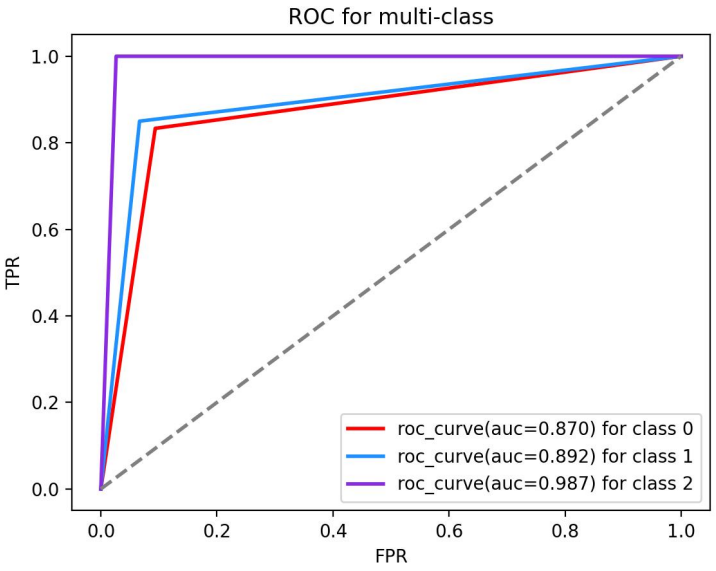
(4)None：返回每个类别的精确率、召回率和 F1-Score，不进行平均。

本次多分类问题中 average 取值为 weighted，即以加权平均的形式进行计算，对于 AUC 的计算则是将各分类进行分别计算。

其他评估指标结果：

Confusion Matrix	30	4	2
	6	34	0
	0	0	34
Precision	0.879		
Accuracy	0.880		
Recall	0.880		
F1_score	0.879		
AUC Class1	0.870		
AUC Class2	0.892		
AUC Class3	0.987		

ROC 曲线：



2. 实验分析

这些指标综合考虑分类模型的不同方面，选择合适的评价指标取决于问题的特性。例如，对于不平衡类别的问题，精确率、召回率和 F1 值可能更具参考价值；而在处理类别不平衡和关注假正例/假负例不同代价的问题时，AUC 值和 ROC 曲线可能更为合适。

四、小结与心得体会

在进行混淆矩阵、准确率、精确率、召回率、F1 值、ROC 曲线和 AUC 值的实验中，我深刻体会到这些评价指标的重要性和应用场景。混淆矩阵提供了详细的分类结果，使我能够深入了解模型在不同类别上的性能。准确率是一个直观的度量，但在不平衡数据中存在局限性，因此精确率和召回率成为更全面评估的关键。F1 值在平衡精确率和召回率方面提供了重要的信息。ROC 曲线和 AUC 值则帮助我理解模型在不同阈值下的性能表现，尤其在选择分类阈值时提供了指导。通过实验，我学到了在不同场景下选择适当的评价指标的重要性，以及如何通过这些指标全面评估和改进分类模型的性能。这次实验深化了我对分类模型评价的理解，为未来更有效地设计和优化模型提供了宝贵经验。

实验八、层次聚类算法设计与应用

一、背景介绍

层次聚类方法对给定的数据集进行层次的分解，直到满足某种条件为止。具体又可分为凝聚的、分裂的两种方案。

- 凝聚的层次聚类是一种自底向上的策略，首先将每个对象作为一个簇，然后合并这些原子簇为越来越大的簇，直到所有的对象都在一个簇中，或者某个终止条件被满足，绝大多数层次聚类方法属于这一类，它们只是在簇间相似度的定义上有所不同。
- 分裂的层次聚类和凝聚的层次聚类相反，采用自顶向下的策略，它首先将所有对象置于一个簇中，然后逐渐细分为越来越小的簇，直到每个对象自成一簇，或者达到了某个终止条件。

二、实验内容

1. 实验要求

下面给出一个样本事务数据库，并对它实施 AGNES 算法和 DIANA 算法，K=2。

序号	属性 1	属性 2	序号	属性 1	属性 2
1	1	1	5	3	4
2	1	2	6	3	5
3	2	1	7	4	4
4	2	2	8	4	5

2. 实验原理

- AGNES 算法：是凝聚的层次聚类方法。AGNES 算法最初将每个对象作为一个簇，然后这些簇根据某些准则被一步步地合并。加入，如果簇C_1中的一个对象和簇C_2中的一个对象之间的距离是所有属于不同簇的对象间欧氏距离最小的，C_1和C_2可能被合并。这是一种单链接方法，其每个簇可以被簇中所有对象代表，两个簇间的相似度由这两个不同簇中距离最近的数据点对的相似度确定。聚类的合并过程反复进行直到所有的对象最终合并形成一个簇。在聚类中，用户能定义希望得到的簇的数目作为一个结束条件。

- DIANA 算法：属于分裂的层次聚类。与凝聚的层次聚类相反，它采用一种自顶向下的策略，它首先将所有对象置于一个簇中，然后逐渐细分为越来越小的簇，直到每个对象自成一簇，或者达到了某个终结条件，例如达到了某个希望的簇数目，或者两个最近簇之间的距离超过了某个阈值。

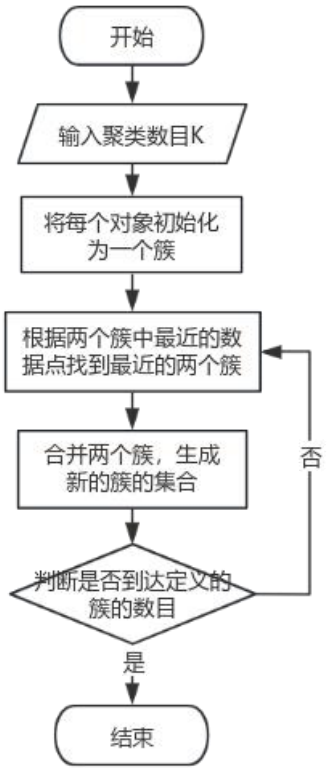
  DIANA 算法使用下面两种测度方法。

  - 簇的直径:在一个簇中的任意两个数据点都有一个欧氏距离，这些距离中的最大值是簇的直径。
  - 平均相异度（平均距离）：

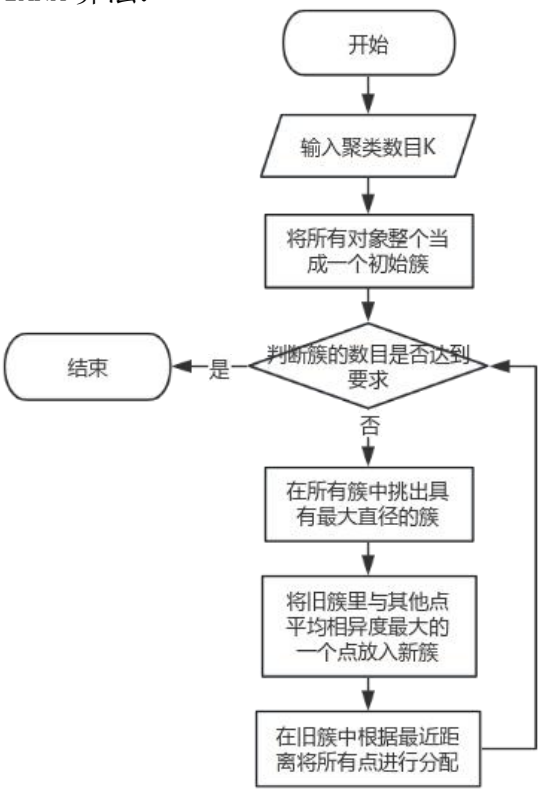
$$d_{avg}(C_i, C_j) = \frac{1}{n_i n_j} \sum_{x \in C_i} \sum_{y \in C_j} |x - y|$$

3. 程序流程图

AGNES 算法：



DIANA 算法：



4. 实验步骤

AGNES 算法：

(1) **确定聚类数目 K：**在进行凝聚层次聚类之前，需要确定簇的数量 K。

(2) **初始化对象：**将样本中的所有对象分别自成一个簇。

(3) **迭代合并簇：**通过以下步骤迭代进行簇的合并。

a) 寻找最近的两个簇：根据两个簇中最近的数据点找到最近的两个簇。

b) 合并簇：合并两个簇，生成新的簇的集合。

(4) **聚类数目检验：**判断算法得出的簇的数目是否达到给定要求。如果满足停止条件，则算法结束，否则返回第 3 步。

DIANA 算法：

(1) **确定聚类数目 K：**在进行分裂层次聚类之前，需要确定簇的数量 K。

(2) **初始化对象：**将样本中的所有对象整个当成一个簇。

(3) **迭代分裂簇：**通过以下步骤迭代进行簇的分裂。

a) 挑出最大直径的簇：根据簇的直径计算公式在簇中挑出具有最大直径的簇。

b) 分裂簇：将旧簇里与其他点平均相异度最大的一个点放入新簇，然后在旧簇中根据最近距离将所有点进行分配。

(4) **聚类数目检验：**判断算法得出的簇的数目是否达到给定要求。如果满足停止条件，则算法结束，否则返回第 3 步。

5. 关键源代码

AGNES 算法，入口参数分别是①数据集 dataSet；②距离公式 dist；③聚类数目 K。

```
    def agnes(dataSet, dist, k):
     clusters = []  # 初始簇
     distance_clusters = []
     # 第一遍遍历，将所有数据单独自成一个簇
     for data in dataSet:
      clusters.append([data])

     # 获取各簇之间的距离矩阵
     for cluster_i in clusters:
      # 初始化单个簇的距离列表
      distance_cluster = []
      # 计算单个簇与其他簇的距离，并存储起来
      for cluster_j in clusters:
       distance_cluster.append(dist(cluster_i, cluster_j))
       distance_cluster.append(np.linalg.norm(np.array(cluster_i) - np.array(cluster_j)))
      # 将计算后的结果保存在所有簇距离列表中
      distance_clusters.append(distance_cluster)

     count = len(dataSet)
     # 合并簇，如果大于 k 说明分组数还没达到要求，直至目标分组数再结束分组
     while count > k:
      # 找出距离最近的两个簇
      cluster_i_index, cluster_j_index, minDistance = findMin(distance_clusters)
      # 将簇 j 添加到簇 i 的列表中，并删除簇 j 的记录，相当于簇的数量-1
      clusters[cluster_i_index].extend(clusters[cluster_j_index])
      clusters.remove(clusters[cluster_j_index])

      # 获取新簇组之间的距离矩阵
      distance_clusters = []
      for cluster_i in clusters:
       distance_cluster = []
       for cluster_j in clusters:
        distance_cluster.append(dist(cluster_i, cluster_j))
       distance_clusters.append(distance_cluster)
      count -= 1
     return clusters
```

DIANA 算法，入口参数分别是①数据集 dataSet；②聚类数目 K。

```
    def diana(data, k):
        m, n = data.shape
        # 初始化所有样本为一个簇
        cls = [data]
        # 记录当前簇的个数
        q = 1
        # 分裂到指定簇数结束
        while q < k:
```

```
            # 找到直径最大的簇及其位置，从最大直径的簇进行分簇
            C, index = diameterMax(cls)
            # 找到平均相异度最大的点作为新的簇一个样本
            max_val = 0
            j = -1
            for i in range(0, C.shape[0]):
                diff = distinct(C[i], np.delete(C, i, axis=0))
                if max_val < diff:
                    j = i
                    max_val = diff
            # 初始化分裂后的两个集合
            cls_new = C[j].reshape((1, n))
            cls_old = np.delete(C, j, axis=0)
            # new 和 old 集合不再变动结束
            while True:
                count = 0  # 标记集合是否更新过
                for i in range(0, cls_old.shape[0]):
                    l = distinct(cls_old[i], cls_new)
                    r = distinct(cls_old[i], np.delete(cls_old, i, axis=0))
                    # 若 old 集合的样本到 new 的最短距离比到自身的最短距离还要小
                    if l < r:
                        count += 1
                        # 更新 old 和 new
                        cls_new = np.concatenate((cls_new, [cls_old[i]]), axis=0)
                        cls_old = np.delete(cls_old, i, axis=0)
                        break

                if count == 0: break
            # 一分为二，弹出最大直径的簇，添加划分后的簇
            cls.pop(index)
            cls.append(cls_new)
            cls.append(cls_old)
            q += 1
        return cls
```

三、实验结果与分析

1. 实验结果

AGNES 算法实验结果：

```
分组1=[[1, 1], [1, 2], [2, 1], [2, 2]]
分组2=[[3, 4], [3, 5], [4, 4], [4, 5]]
```

DIANA 算法实验结果：

```
分组1=[[1, 1], [1, 2], [2, 1], [2, 2]]
分组2=[[3, 4], [3, 5], [4, 4], [4, 5]]
```

2. 实验分析

(1) AGNES 算法和 DIANA 算法的性能分析

AGNES 算法比较简单，但经常会遇到合并点选择的困难。这样的决定是非常关键的，因为一旦一组对象被合并，下一步的处理将在新生成的簇上进行。已做的处理不能撤销，聚类之间也不能交换对象。如果在某一步没有很好地合并选择，可能会导致低质量的聚类结果。而且，这种聚类方法不具有很好的可伸缩性，因为合并的决定需要检查和估算大量的对象或簇。

DIANA 算法具有直观的自底向上划分方式，适用于处理任意形状的簇结构。然而，由于自底向上的策略，算法在处理大规模数据集时可能面临较高的时间复杂度。DIANA 对异常值较为敏感，可能导致生成不稳定的簇结构。总体而言，DIANA 算法适用于中小规模、非凸形状簇结构的数据集，但在大规模数据和对异常值敏感的场景中，可能需要谨慎选择或结合其他聚类方法。

(2) AGNES 算法和 DIANA 算法的异同

- 相同点
  - ✧ AGNES 和 DIANA 都属于层次聚类算法，这意味着它们通过递归地划分或合并簇，构建一个层次结构，形成一个聚类树（或树状图）。
  - ✧ 都需要定义簇间的距离度量或相似性度量，以确定合并或分裂的标准。
  - ✧ AGNES 和 DIANA 都能够形成一个树状结构，其中每个节点代表一个簇，边表示合并或分裂的过程。
- 不同点
  - ✧ AGNES 采用自底向上的策略，DIANA 采用自顶向下的策略。
  - ✧ AGNES 的计算复杂度相对较高，因为在每一步都需计算所有簇对之间的距离。DIANA 在初始时计算较少的距离，但递归过程中可能需要计算的距离逐渐增多。
  - ✧ AGNES 对于缺失值较为敏感，因为在计算簇间距离时，需要考虑所有数据点的距离。DIANA 对于缺失值的影响相对较小，因为它主要关注簇内的距离。

四、小结与心得体会

层次聚类算法是一类将数据集划分为层次结构的聚类方法，其中 AGNES 和 DIANA 是两种典型的层次聚类算法。AGNES 算法采用自底向上的策略，首先将每个数据点视为一个簇，然后迭代地将相似度最高的簇合并，形成更大的簇，直到达到停止条件。这个过程构建了一个树形簇结构。AGNES 的优点包括简单易理解、不需预先指定簇的数量，但其计算复杂度较高，尤其是对于大规模数据集。相反，DIANA 算法采用自顶向下的策略，首先将整个数据集视为一个簇，然后迭代地将当前簇分裂为更小的簇，直到满足停止条件。DIANA 的优点在于对异常值较为鲁棒，但可能生成不稳定的簇结构，且对大规模数据的计算开销较高。在实践中，我发现 AGNES 更适合处理相对规模较小且簇结构相对紧凑的数据集。其直观的自底向上合并方式有助于捕捉整体相似性。另一方面，DIANA 适用于数据集中存在松散簇的情况，而其自顶向下的分裂方式对于探索潜在的子簇结构更为有利。

总体而言，层次聚类算法具有直观性和不需要预先指定簇数的优势，但在大规模数据和计算效率方面可能受到挑战。在选择 AGNES 或 DIANA 时，需要根据数据的特点和问题需求进行权衡。此外，这两种算法的性能也受到距离度量的选择和合并或分裂策略的影响，需要谨慎调整参数以获得更好的聚类效果。

实验九、朴素贝叶斯算法设计与应用

一、背景介绍

贝叶斯分类是统计分类的方法。在贝叶斯学习方法中实用性很高的一种称为朴素贝叶斯分类方法。在某些领域，其性能与神经网咯和决策树相当。

二、实验内容

1. 实验要求

请对下表中的训练数据，使用朴素贝叶斯方法进行分类学习，并对未知样本 $\boldsymbol{X} = (age = " \leq 30", income = "medium", student = "yes", credit\_rating = "fair")$，进行分类预测。

RID	Age	Income	Student	Credit_rating	Buys_computer
1	≤30	High	No	Fair	No
2	≤30	High	No	Excellent	No
3	31~40	High	No	Fair	Yes
4	＞40	Medium	No	Fair	Yes
5	＞40	Low	Yes	Fair	Yes
6	＞40	Low	Yes	Excellent	No
7	31~40	Low	Yes	Excellent	Yes
8	≤30	Medium	No	Fair	No
9	≤30	Low	Yes	Fair	Yes
10	＞40	Medium	Yes	Fair	Yes
11	≤30	Medium	Yes	Excellent	Yes
12	31~40	Medium	No	Excellent	Yes
13	31~40	High	Yes	Fair	Yes
14	＞40	Medium	No	Excellent	No

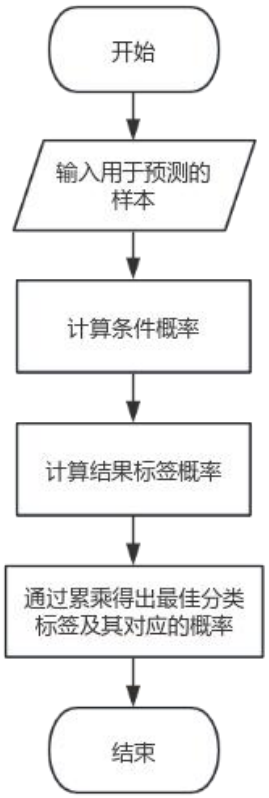
2. 实验原理

(1) 贝叶斯定理：贝叶斯定理是基于条件概率的一种概率推断方法。对于分类问题，它可以表示为$P(C|X) = \frac{P(X|C)*P(C)}{P(X)}$，其中，$P(C|X)$是给定观测数据 X 时类别 C 的概率，$P(X|C)$是在类别 C 下观测数据 X 的概率，$P(C)$是类别 C 的先验概率，$P(X)$是观测数据 X 的概率。

(2) 朴素假设：朴素贝叶斯算法假设特征之间相互独立，即给定类别条件下，特征之间的关系是独立的。这个假设使得计算条件概率变得简单，从而简化了模型。

(3) 分类决策：在进行分类时，朴素贝叶斯算法选择具有最大条件概率的类别作为预测结果。即对于新观测数据 X，计算每个类别的后验概率，然后选择概率最大的类别。

3. 程序流程图



4. 实验步骤

(1) **输入预测样本：**将未知分类样本的输入数据输入。
(2) **计算条件概率：**计算在各结果情况下，给定未知样本数据中各属性的条件概率。
(3) **计算结果标签概率：**计算在各结果的概率值。
(4) **累乘预测结果：**将各结果标签概率与其对应的条件概率进行累乘，选择最大的条件概率的类别作为预测结果，并输入概率值和分类结果。

5. 关键源代码

(1) 计算条件概率，入口参数为分别是①数据集 dataSet；②测试数据的条件列表 list_condition；③标签数量的数量集 label_count。

```
1.     def calcCondProb(dataSet, list_condition, label_count):  # list_condition 是测试数据的条件列表，label_count 是标签数量
2.         print('--------------条件概率数据-----------------')
3.         dict_condProb = {}  # 用于存放条件概率的字典
4.         # 通过遍历标签次数字典，获取对应条件的概率
5.         for label, count in label_count.items():
6.             subDataSet = splitDataSet(dataSet, label)  # 获得分割后的数据集子集，例如，label 为yes，则splitDataSet 里面就是标签为yes 的数据
7.             # 遍历给定的条件列表，得出条件概率
8.             for i in range(len(list_condition)):
9.                 # 通过numpy 将subDataSet 转为数组，获取第i 列，即给定条件对应的列，然后获取该条件的数据数量
10.                conditionCount = list(np.array(subDataSet)[:, i]).count(list_condition[i])
```

```
                condProb = conditionCount/count  # 计算在标签结果下的该条件的条件概率
                string_condition = list_condition[i] + "|" + label  # 用于辨识条件概率的键值，格式为“xxx|xxx”
                print("P(%s)=%.2f" % (string_condition, condProb))
                dict_condProb[string_condition] = condProb
        return dict_condProb
```

(2)分类和预测函数，入口参数为给定的条件列表 list_condition。

```
    def classifyAndPredict(list_condition):
        dataSet = creatDataSet()  # 创建数据集
        label_count = calcLabelCount(dataSet)   # 计算结果标签数量
        print(f'各标签结果对应数量={label_count}')
        dict_condProb = calcCondProb(dataSet, list_condition, label_count)  # 计算条件概率
        dict_labelProb = calcLabelProb(label_count)  # 计算标签概率
        predictLabel = "无"
        maxProb = 0.0
        # 通过遍历累乘得出最佳分类标签以及其对应的概率。
        for label, labelProb in dict_labelProb.items():
            resultProb = labelProb
            for condition, condProb in dict_condProb.items():
                if label in condition[-3:]:
                    resultProb *= condProb
            if resultProb > maxProb:
                maxProb = resultProb
                predictLabel = label
        return maxProb, predictLabel
```

三、实验结果与分析

1. 实验结果

```
各标签结果对应数量={'no': 5, 'yes': 9}
--------------条件概率数据-----------------
P(young|no)=0.60
P(medium|no)=0.40
P(yes|no)=0.20
P(fair|no)=0.40
P(young|yes)=0.22
P(medium|yes)=0.44
P(yes|yes)=0.67
P(fair|yes)=0.67
-----------------预测结果------------------
age,income,student,credit_rating=['young', 'medium', 'yes', 'fair'] ===>>> 分类预测:yes + 预测概率:0.028
```

2. 实验分析

朴素贝叶斯算法的性能分析：

首先，朴素贝叶斯算法具有良好的可解释性和计算效率，特别适用于高维度数据集和大规模文本分类。其训练和预测过程简单，对于小规模数据集表现良好。然而，朴素贝叶斯算法依赖于朴素假设，即特征之间相互独立，这在实际数据中不总是成立，可能导致模型对特征之间的依赖关系估计不准确。此外，在类别不平衡或存在噪声的情况下，朴素贝叶斯可能表现较差。综合而言，朴素贝叶斯算法在特定场景下表现优越，但在某些假设条件不满足的情况下，性能可能受到限制。

四、小结与心得体会

朴素贝叶斯算法是一种基于概率统计的分类算法，通过对特征条件独立性的假设，简化了模型的计算过程。朴素贝叶斯对于文本分类等高维数据场景表现出色，尤其在数据较小且特征之间相对独立的情况下，其训练和预测速度快，且对噪声和异常值具有一定的鲁棒性。然而，朴素贝叶斯的性能受到朴素假设的限制，特征之间的真实关系可能被忽略。在实际应用中，合适的特征工程和平滑技术能够改善模型性能。对于本次实验实践，仍存在一些改进的空间，例如可以引入平滑技术来处理概率为零的情况，进而提高模型的鲁棒性。对于测试数据集的条件概率计算，可以通过更简洁的方式实现，提高代码的可读性。总体而言，朴素贝叶斯算法是一种简单而有效的分类工具，在特定场景下能够取得令人满意的结果。对于代码的实现也有待改善。