商务智能-课后作业2

1 请描述数据挖掘的步骤，请简述每个步骤的基本含义。

（1）数据清理：清理数据，指从数据库或数据表中更正和删除不准确数据记录，去除噪声和不一致的数据的过程。通过有效的数据清理，能够确保所有数据集应保持一致并且没有任何错误，为以后数据的使用和分析提供支撑。

（2）数据集成：多种数据源的融合，即将来自众多异构数据源的数据合并成一致的数据，以保留和支持统一的观点的信息。

（3）数据选择：从数据库中提取和分析相关的数据，形成数据集，一般包括训练集、验证集、测试集的数据。

（4）数据转换：将数据从一种格式或结构转换为另一种格式或结构的过程，数据转换对于数据集成和数据管理等活动至关重要，能把数据变换和统一成适用于挖掘的形式。

（5）数据挖掘：使用智能方法，根据数据仓库中的数据信息，选择合适的分析工具，应用统计方法、事例推理、决策树、规则推理、模糊集、甚至神经网络、遗传算法的方法处理信息，得出有用的分析信息，提取数据模式。

（6）模式评估：从商业角度，根据某种兴趣度度量，识别代表知识的真正有趣的模式，由行业专家来验证数据挖掘结果的正确性。

（7）知识表示：将数据挖掘所得到的分析信息以可视化的方式呈现给用户，或作为新的知识存放在知识库中，供其他应用程序使用。

2 请编写程序实现Apriori算法和FP-Growth算法，算法可以根据给定的支持度和置信度获取所有的频繁项集和关联规则。请自拟数据集进行测试。

拟定数据集如下：

```
dataSet =
    [('尿布', '啤酒', '奶粉', '洋葱'),
     ('尿布', '啤酒', '奶粉', '洋葱'),
     ('尿布', '啤酒', '苹果', '洋葱'),
     ('尿布', '啤酒', '苹果'),
     ('尿布', '啤酒', '奶粉'),
     ('尿布', '啤酒', '奶粉'),
     ('尿布', '啤酒', '苹果'),
     ('尿布', '啤酒', '苹果'),
     ('尿布', '奶粉', '洋葱'),
     ('奶粉', '洋葱')]
```

（1）Apriori 算法

Apriori 算法的核心一是找出频繁项集，二是从频繁项集中提取关联规则，具体步骤如下：

1. 生成 k = 1 个元素的项集列表
2. 扫描数据集，裁剪不满足最小支持度的项集。剪枝的重要依据是，只有子集都是频繁项集的项集才是频繁项集。
3. 对余下项集进行组合，生成 k + 1 个元素的项集

4. 重复步骤2和3
5. 返回所有频繁项集的列表
6. 由频繁项集产生关联规则

程序件 apriori.py 是不调用类库直接实现的 Apriori 算法：

函数 `create_C1` 生成 k = 1 个元素的项集列表，函数 `create_Ck` 生成 k + 1 个元素的项集

函数 `is_apriori` 根据最小支持度剪枝：

```
def is_apriori(Ck_item, Lk):
    for item in Ck_item:
        sub_Ck = Ck_item - frozenset([item])
        if sub_Ck not in Lk:
            return False
    return True
```

函数 `generate_rules` 根据最小置信度生成关联规则：

```
def generate_rules(L, minConf):
    big_rule_list = []
    sub_set_list = []
    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):
                    support = supportData[freq_set] / supportData[freq_set -
sub_set]
                    big_rule = (freq_set - sub_set, sub_set, support)
                    if support >= minConf and big_rule not in big_rule_list:
                        big_rule_list.append(big_rule)
            sub_set_list.append(freq_set)
    return big_rule_list
```

测试用例中，设置最小支持度 minSupport = 0.6 ，最小置信度 minConf = 1 。得到如下输出结果：

```
------------------------------------------------------------
元素数 k=1
------------------------------------------------------------
频繁项集: {'尿布'}  支持度: 0.9
频繁项集: {'啤酒'}  支持度: 0.8
频繁项集: {'奶粉'}  支持度: 0.6
------------------------------------------------------------
元素数 k=2
------------------------------------------------------------
频繁项集: {'啤酒', '尿布'}  支持度: 0.8
------------------------------------------------------------
关联规则
{'啤酒'}  --->  {'尿布'}  置信度: 1.0
```

分析可知，当最小支持度 minSupport = 0.6 ，最小置信度 minConf = 1 时，k = 1 个元素的项集列表有 `{'尿布'}, {'啤酒'}, {'奶粉'}` 三个，k = 2 个元素的项集列表有 `{'啤酒', '尿布'}` 一个。根据得到的频繁项集，产生的关联规则为 `{'啤酒'} ---> {'尿布'}`，且其置信度为 `1.0` 。

程序 apriori_lib.py 是调用类库 efficient_apriori 实现的 Apriori 算法。

调用函数 `apriori()`，即可得到频繁项集 itemsets 和关联规则 rules：

```
itemsets, rules = apriori(data, min_support=0.6, min_confidence=1)
```

其中，`apriori()` 函数的构造形式为：

```
apriori(transactions: Iterable[Union[set, tuple, list]], min_support: float =
0.5,
        min_confidence: float = 0.5, max_length: int = 8, verbosity: int = 0,
        output_transaction_ids: bool = False)
```

测试用例中，设置最小支持度 minSupport = 0.6 ，最小置信度 minConf = 1 。得到如下输出结果：

```
--------------------------------------------------
频繁项集:
{1: {('尿布',): 9, ('啤酒',): 8, ('奶粉',): 6}, 2: {('啤酒', '尿布'): 8}}
--------------------------------------------------
规则:
[{啤酒} -> {尿布}]
--------------------------------------------------
```

(2) FP-Growth 算法

FP-Growth 算法通过构造一个树结构来压缩数据记录，使得挖掘频繁项集只需要扫描两次数据记录，而且该算法不需要生成候选集合，所以效率会比较高。FP-Growth 算法的具体步骤如下：

1. 遍历数据集，统计各元素项出现次数，创建头指针表
2. 移除头指针表中不满足最小值尺度的元素项
3. 第二次遍历数据集，创建FP树。对每个数据集中的项集：
    1. 初始化空FP树
    2. 对每个项集进行过滤和重排序
    3. 使用这个项集更新FP树，从FP树的根节点开始：
        1. 如果当前项集的第一个元素项存在于FP树当前节点的子节点中，则更新这个子节点的计数值
        2. 否则，创建新的子节点，更新头指针表
        3. 对当前项集的其余元素项和当前元素项的对应子节点递归过程

程序 fp-growth.py 是不调用类库直接实现的 FP-Growth 算法：

函数 `countitem` 遍历数据进行计数，函数 `deletekey` 删除支持度不够的 key

函数 `sorfarray` ，删除支持度不够的元素，并得到从大到小排序的数组：

```
def sorfarray(array, dict, delect):
    newarray = []
    # 删除支持度不够的元素
```

```
    for item in array:
        temp = {}
        for value in item:
            if value in delect:
                pass
            else:
                temp[value] = dict[value]
        temp = sorted(temp.items(), key=lambda d: d[1], reverse=True)
        tem = []
        for tuple in temp:
            tem.append(tuple[0])
        newarray.append(tem)
    return newarray
```

函数 `confidence`，得到每个种类的置信度：

```
def confidence(alldict, support, newinfo):
    newdict = {}
    for kind in alldict:
        copydict = alldict[kind].copy()
        for key in alldict[kind]:
            if alldict[kind][key] < support:
                copydict.pop(key)
        if copydict:
            newdict[kind] = copydict

    # 计算置信度
    for kind in newdict:
        for key in newdict[kind].keys():
            tempnum = newdict[kind][key]
            denominator = 0
            for item in newinfo:
                if len(key) == 1:
                    if key in item:
                        denominator += 1
                elif len(key) == 2:
                    if key[0] in item and key[1] in item:
                        denominator += 1
                elif len(key) == 3:
                    if key[0] in item and key[1] in item and key[2] in item:
                        denominator += 1

            newdict[kind][key] = str(tempnum) + "/" + str(denominator)
    return newdict
```

测试用例中，设置 `support = 3`，得到如下结果：

```
--------------------------------------------------
dict
{'尿布': 9, '啤酒': 8, '奶粉': 6, '洋葱': 5, '苹果': 4}
--------------------------------------------------
alldict
{'啤酒': {'尿布': 8}, '奶粉': {'尿布': 5, '啤酒': 4, '尿布啤酒': 4}, '尿布': {}, '洋
葱': {'尿布': 4, '啤酒': 3, '奶粉': 4, '尿布啤酒奶粉': 2, '尿布啤酒': 1, '尿布奶粉':
1}, '苹果': {' 尿布': 4, '啤酒': 4, '洋葱': 1, '尿布啤酒洋葱': 1, '尿布啤酒': 3}}
--------------------------------------------------
confidence
{'啤酒': {'尿布': '8/0'}, '奶粉': {'尿布': '5/0', '啤酒': '4/0', '尿布啤酒': '4/0'},
'洋葱': {'尿布': '4/0', '啤酒': '3/0', '奶粉': '4/0'}, '苹果': {'尿布': '4/0', '啤
酒': '4/0', '尿布
啤酒': '3/0'}}
--------------------------------------------------
```

其中，`dict` 表示各项在数据集中出现的次数，`alldict` 表示所有组合的字典，`confidence` 为每个种类的置信度。

程序 fp-growth_lib.py 是调用类库 mlxtend.frequent_patterns 实现的 FP-Growth 算法：

调用函数 `fpgrowth` ，即可得到所有的频繁项集：

```
frequent_itemsets = fpgrowth(df, min_support=0.8, use_colnames=True)
```

调用函数 `association_rules`，即可得到关联规则：

```
rules = association_rules(frequent_itemsets, metric="confidence",
min_threshold=1)
```

测试用例中，设置 `min_support=0.8` ， `min_threshold=1`，得到如下输出结果：

```
频繁项集：
     support   itemsets
0        0.9      (尿布)
1        0.8      (啤酒)
2        0.8      (尿布, 啤酒)
关联规则：
  antecedents consequents  antecedent support  consequent support  support
confidence      lift  leverage  conviction
0   (啤酒)   (尿布)   0.8   0.9   0.8   1.0   1.11   0.08   inf
```

3 请编写程序实现决策树算法，请自拟数据集进行测试。

决策树（Decision Tree）是有监督学习中的一种算法，并且是一种基本的分类与回归的方法。构建一颗决策树的大致步骤如下：

1. 导入需要的算法库和模块以及数据集
2. 实例化数据集
3. 查看数据集属性

4. 划分数据
5. 建立模型并训练模型
6. 计算准确度
7. 查看特征的重要性

（1）程序 tree.py 是不调用类库直接实现的决策树构建

自拟数据集如下：

```
# 数据集
dataSet = [
    ['奥地利', '10', '红酒'],
    ['荷兰', '10', '红酒'],
    ['荷兰', '10', '红酒'],
    ['奥地利', '5', '白酒'],
    ['荷兰', '5', '白酒'],
    ['荷兰', '10', '白酒'],
    ['奥地利', '10', '白酒'],
    ['奥地利', '10', '白酒']]

#特征
labels = ['产地', '年份']
```

函数 `calcEntropy` 计算熵，函数 `splitDataSet` 划分数据集

函数 `chooseBestFeatureToSplit` 根据信息增熵，选择最优的分类特征：

```
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1
    baseEntropy = calcEntropy(dataSet)
    bestInfoGain = 0
    bestFeatIndex = -1
    for i in range(numFeatures):
        featList = [row[i] for row in dataSet]
        uniqueFeatValues = set(featList)  # 用集合去重，得到特征值

        newEntropy = 0
        for value in uniqueFeatValues:  # 用特征值中的每一个 划分数据集
            subDataSet = splitDataSet(dataSet, i, value)
            print('子集 :', subDataSet)
            weight = len(subDataSet) / float(len(dataSet))  # 权重，子集个数/ 全集
个数
            newEntropy += weight * calcEntropy(subDataSet)
        print('信息熵 :', newEntropy)
        infoGain = baseEntropy - newEntropy  # 信息增益
        if infoGain > bestInfoGain:
            bestInfoGain = infoGain
            bestFeatIndex = i
    return bestFeatIndex
```

函数 `createTree` 构建决策树：

```
def createTree(dataSet, labels):
    typeList = [row[-1] for row in dataSet]  # 类别
    if typeList.count(typeList[0]) == len(typeList):
```

```
        return typeList[0]
    if len(dataSet[0]) == 1:
        return majorityCnt(typeList)
    bestFeatIndex = chooseBestFeatureToSplit(dataSet)  # 最优特征下标和对应特征
    bestFeat = labels[bestFeatIndex]
    print('bestFeatureIndex =', bestFeatIndex)
    print('--------------', bestFeat, end='--------------\n')

    myTree = {bestFeat: {}}  # 分类结果以字典形式保存
    del (labels[bestFeatIndex])

    uniqueVals = set()  # 最优特征能取的值
    {uniqueVals.add(row[bestFeatIndex]) for row in dataSet}
    print(f'{bestFeat} 取值 :', uniqueVals)
    for value in uniqueVals:
        subLabels = labels
        myTree[bestFeat][value] = createTree(splitDataSet(dataSet,
bestFeatIndex, value), subLabels)
    return myTree
```

测试程序输出如下：

```
特征数 = 3
样本数 {'红酒': 3, '白酒': 5}
entropy = 0.9544340029249649
子集 : [['10', '红酒'], ['10', '红酒'], ['5', '白酒'], ['10', '白酒']]
特征数 = 2
样本数 {'红酒': 2, '白酒': 2}
entropy = 1.0
子集 : [['10', '红酒'], ['5', '白酒'], ['10', '白酒'], ['10', '白酒']]
特征数 = 2
样本数 {'红酒': 1, '白酒': 3}
entropy = 0.8112781244591328
信息熵 : 0.9056390622295665
子集 : [['奥地利', '红酒'], ['荷兰', '红酒'], ['荷兰', '红酒'], ['荷兰', '白酒'], ['奥
地利', '白酒'], ['奥地利', '白酒']]
特征数 = 2
样本数 {'红酒': 3, '白酒': 3}
entropy = 1.0
子集 : [['奥地利', '白酒'], ['荷兰', '白酒']]
特征数 = 2
样本数 {'白酒': 2}
entropy = 0.0
信息熵 : 0.75
bestFeatureIndex = 1
-------------- 年份--------------
年份 取值 : {'10', '5'}
特征数 = 2
样本数 {'红酒': 3, '白酒': 3}
entropy = 1.0
子集 : [['红酒'], ['红酒'], ['白酒']]
特征数 = 1
样本数 {'红酒': 2, '白酒': 1}
entropy = 0.9182958340544896
子集 : [['红酒'], ['白酒'], ['白酒']]
特征数 = 1
样本数 {'红酒': 1, '白酒': 2}
```

```
entropy = 0.9182958340544896
信息熵 : 0.9182958340544896
bestFeatureIndex = 0
-------------- 产地--------------
产地 取值 : {'荷兰', '奥地利'}
typeCount = {'红酒': 2, '白酒': 1}
多数为 : 红酒
typeCount = {'红酒': 1, '白酒': 2}
多数为 : 白酒

{'年份': {'10': {'产地': {'荷兰': '红酒', '奥地利': '白酒'}}, '5': '白酒'}}
```

输出结果包括了完整的样本数计算和子集的划分。最后一行即为构建的决策树。

(2) **程序 tree_lib.py 调用 sklearn.tree 实现决策树的构建，计算准确度和特征重要性。**

数据集选用 sklearn.datasets 中的红酒数据集。红酒数据集的特征矩阵包括 178 个数据项，13 个特征点。通过分析 13 个特征点来对红酒进行分类。

调用函数 `load_wine`，实例化红酒数据集，并得到特征矩阵：

```
wine = load_wine()
print(wine.data.shape)
```

调用函数 `train_test_split`，将数据集划分为测试集和训练集：

```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(wine.data, wine.target,
test_size=0.3)
```

调用函数 `DecisionTreeClassifier`，建立模型并训练模型：

```
clf = tree.DecisionTreeClassifier(criterion="entropy")
clf = clf.fit(Xtrain, Ytrain)
```

测试程序中，将数据集的30%划分为测试集，其他的划分为训练集，输出结果如下：

```
--------------------------------------------------
shape
(178, 13)
--------------------------------------------------
score
0.9814814814814815
--------------------------------------------------
feature importances
[0.02568722 0.         0.         0.01044536 0.         0.
 0.41252758 0.         0.         0.21665416 0.         0.
 0.33468568]
--------------------------------------------------
```

其中，`shape` 为红酒数据集的特征矩阵，`score` 为准确度得分，`feature importances` 为特征重要性。

（3）**python 文件 tree_lib_plt.py 调用 matplotlib.pyplot 实现决策树的可视化。**

数据集选用 sklearn.datasets 中的红酒数据集。红酒数据集的特征矩阵包括 178 个数据项，13 个特征点。通过分析 13 个特征点来对红酒进行分类。

调用函数 `DecisionTreeClassifier`，建立模型并训练模型，限制树的最大深度为 4：

```
clf = DecisionTreeClassifier(max_depth=4)
clf.fit(X, y)
```

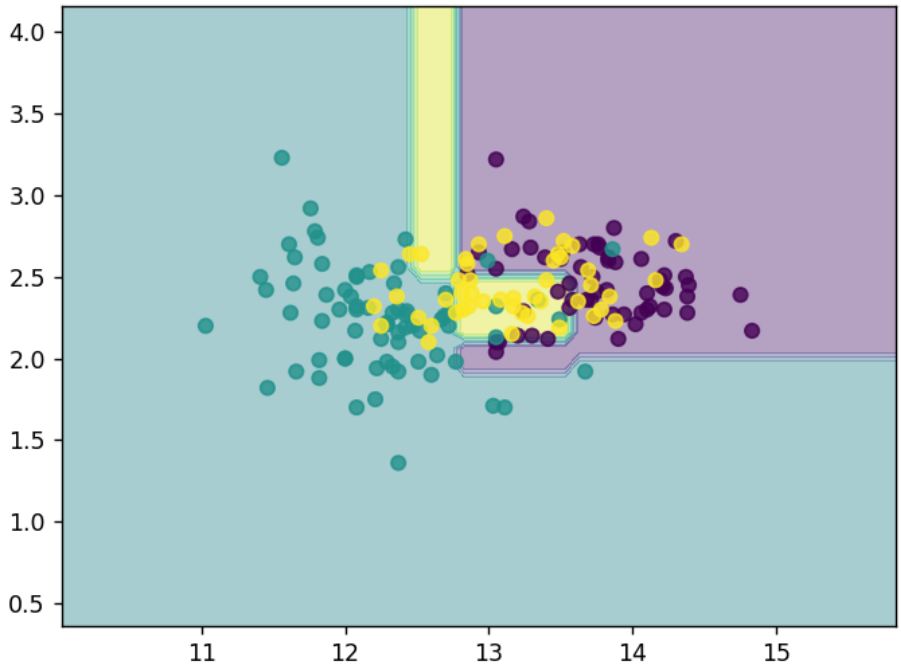
调用类库 `numpy` 和 `matplotlib.pyplot` 完成决策树的绘制：

```
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))

Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.4)
plt.scatter(X[:, 0], X[:, 1], c=y, alpha=0.8)
plt.show()
```

测试程序输出如下图所示，可以看到，数据被大致分为三类：



4 请给出贝叶斯分类的伪代码，并说明该算法是如何从后验概率的计算变化为容易计算的形式的。

（1）贝叶斯分类算法的基本原理：

1. 将训练和分类的文档一致处理为词向量。
2. 通过贝叶斯算法对数据集进行训练，统计所有词向量各种分类的概率。
3. 对于待分类的文档，在转换为词向量之后，从训练集中取得该词向量为各种分类的概率，概率最大的分类就是所求分类结果。

(2) 贝叶斯分类的伪代码：

计算条件概率：

```
对每篇训练文档：
    对每个类别：
        增加该单词计数值
        增加所有单词计数值
    对每个类别：
        对每个单词：
            将该单词的数目除以单词总数得到条件概率
返回所有单词在各个类别下的条件概率
```

贝叶斯分类：

```
获取数据
数据处理：
    类别计数
    统计文本长度
训练和预测：
    构建文本的向量
    计算条件概率
    进行预测
```

贝叶斯分类器训练函数的代码实现：

```python
def trainNB(trainMatrix, trainCategory):
    numTrainDocs = len(trainMatrix)
    numWords = len(trainMatrix[0])
    pAbusive = sum(trainCategory)/float(numTrainDocs)
    p0Num = ones(numWords);
    p1Num = ones(numWords);    # 用来统计两类数据中，各词的词频
    p0Denom = 2.0;  # 用于统计0类中的总数
    p1Denom = 2.0  # 用于统计1类中的总数
    for i in range(numTrainDocs):
        if trainCategory[i] == 1:
            p1Num += trainMatrix[i]
            p1Denom += sum(trainMatrix[i])
        else:
            p0Num += trainMatrix[i]
            p0Denom += sum(trainMatrix[i])
    p1Vect = log(p1Num / p1Denom)     # 在类1中，每个词的发生概率
    p0Vect = log(p0Num / p0Denom)
    return p0Vect, p1Vect, pAbusive
```

(3) 从后验概率的计算变化为容易计算的形式

贝叶斯公式：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

此公式将后验概率，即事件 A 在事件 B 发生的条件下的概率 P(A|B) ，转化为求条件概率 P(B|A) 即事件 B 在事件 A 发生的条件下的概率，A 的先验概率 P(A) 和 B 的先验概率 P(B) 。此时，先验概率 P(A) 和 P(B) 可以直接通过数据分析求得，条件概率 P(B|A) 也可以根据现有的数据结果得到。

简单来说，贝叶斯公式，就是在知道结果的情况下去推断原因的方法。通过现象和结果去推断事情发生的原因，把后验概率的计算变化为容易计算的形式。

5 请分析k-means算法不适合的场景。

（1）用户难以直接给出簇数 K：

进行k-means算法时，必须指定聚类数量。但是有时候我们并不知道应该聚成多少个类，而是希望算法可以给出一个合理的聚类数量，往往一开始k值很难预先估计并给定。如果某种场景中，用户难以直接给出簇数 K，那么不适合的 K 值可能返回较差的结果。

（2）非凸形状的簇或是大小差别很大的簇：

K-means 聚类算法的核心思想是不断调整中心点来计算距离，然后生成新的中心点，直到达到平衡为止。即随机选取k个点作为聚类中心，计算其他点与中心点的距离，选择距离最近的中心并归类，归类完成后计算每类的新中心点，重新计算每个点与中心点的聚类并选择距离最近的归类，重复此过程，直到中心点不再变化。因此 K-means 只适合聚类那些依照距离来分布的数据，而对于非凸形状的簇或是大小差别很大的簇，有效更新簇中心。

（3）簇中噪声和离群点过多：

K-means 在聚类过程中同等的看待每个特征维度，因此当数据集中存在噪音维度时，Kmeans 在聚类过程中仍旧将其看成是正常的特征维度进行利用，而不能加以区分，致使簇中心向噪声和离群点偏离。

6 请使用单连接算法描述下列数据是如何进行层次聚类的并画出树状图。

（10，8） （70，80） （99，87） （6，5） （5，10）。

将五个样本都分别看成是一个簇，依次标为 1、2、3、4、5 。最靠近的两个簇是 1 和 4 ，因为他们具有最小的簇间距离 D（1，4） = 5.00

	x_1	x_2
1	10	8
2	70	80
3	99	87
4	6	5
5	5	10

	1	2	3	4	5
1	0.00				
2	93.72	0.00			
3	119.00	29.83	0.00		
4	5.00	98.00	123.99	0.00	
5	5.39	95.52	121.51	5.10	0.00

Step1：合并簇 1 和 4 ，得到新簇集合 （14）、2、3、5，更新距离矩阵：

D（2，（14））= min（D（2，1），D（2，4））= min（93.72，98.60）= 93.72

D（3，（14））= min（D（3，1），D（3，4））= min（119.00，123.99）= 119.00

D（5，（14））= min（D（5，1），D（5，4））= min（5.39，5.10）= 5.10

原有簇 2、3、5 间的距离不变，修改后的距离矩阵如下图所示。最靠近的两个簇是 （14） 和 5 ，因为他们具有最小的簇间距离 D（（14），5） = 5.10

Step2：合并簇 （14） 和 5 ，得到新簇集合 （145）、2、3，更新距离矩阵：

D（2，（145））= min（D（2，（14）），D（2，5））= min（93.72，95.52）= 93.72

D（3，（145））= min（D（3，（14）），D（3，5））= min（119.00，121.51）= 119.00

原有簇 2、3 间的距离不变，修改后的距离矩阵如下图所示。最靠近的两个簇是 2 和 3 ，因为他们具有最小的簇间距离 D（2，3） = 29.83

Step3：合并簇 2 和 3 ，得到新簇集合 （145）、（23），更新距离矩阵：

D（（23），（145））= min（D（2，（145）），D（3，（145））） = min（93.72，119.00）= 93.72

修改后的距离矩阵如下图所示。最靠近的两个簇是 （145） 和 （23） ，因为他们具有最小的簇间距离 D（（145），（23）） = 93.72

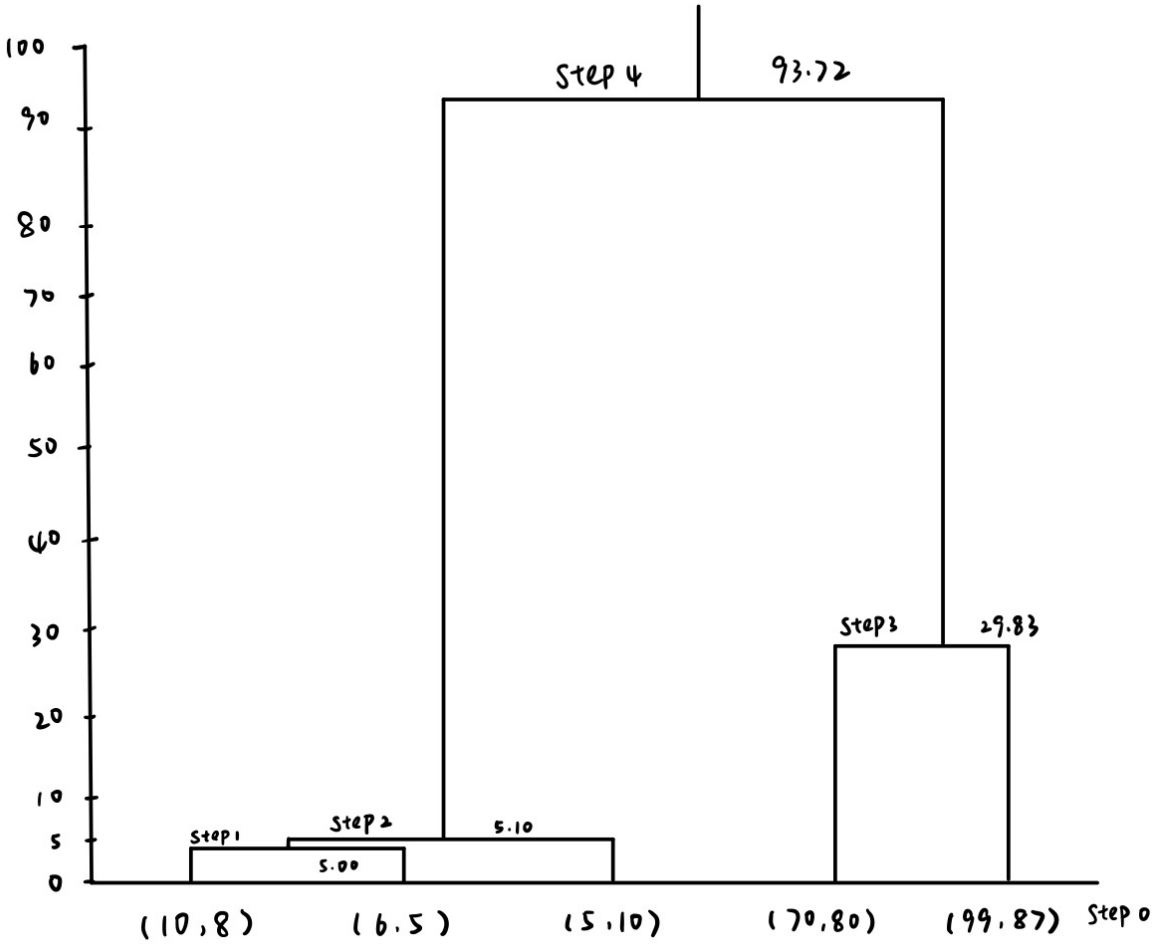
Step4：合并簇 （145） 和 （23）

	(14)	2	3	5
(14)	0.00			
2	93.72	0.00		
3	119.00	29.83	0.00	
5	5.10	95.52	121.51	0.00

	(145)	2	3
(145)	0.00		
2	93.72	0.00	
3	119.00	29.83	0.00

	(145)	(23)
(145)	0.00	
(23)	93.72	0.00

最终得到单连接树图如下：


100
90
80
70
60
50
40
30
20
10
5
0
Step 4
93.72
Step3
29.83
Step 2
5.10
Step 1
5.00
(10,8)
(6.5)
(5,10)
(70,80)
(99,87)
Step 0