电子科技大学

计算机科学与工程学院

标 准 实 验 报 告

（实验）课程名称 人工智能

电子科技大学教务处制表

电 子 科 技 大 学

电 子 科 技 大 学

实 验 报 告

学生姓名：李天　　学 号：2020080904021

指导教师：段立新 张彦如 顾实

实验地点： 主楼 A2-413-1　　实验时间：2022.12.3

一、实验室名称：计算机学院实验中心

二、实验项目名称：决策树实验

三、实验学时：5 学时

四、实验原理：

ID3 算法的核心思想就是以信息增益度量属性选择，选择分裂后信息增益最大的属性进行分裂。下面先定义几个要用到的概念。设 D 为用类别对训练元组进行的划分，则 D 的熵（entropy）表示为：

$$\inf o(D) = -\sum_{i=1}^{m} p_i \log_2(p_i) \qquad \text{（公式 1-1）}$$

其中 pi 表示第 i 个类别在整个训练元组中出现的概率，可以用属于此类别元素的数量除以训练元组元素总数量作为估计。熵的实际意义表示是 D 中元组的类标号所需要的平均信息量。现在我们假设将训练元组 D 按属性 A 进行划分，则 A 对 D 划分的期望信息为：

$$\inf o_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \inf o(D_j) \qquad \text{（公式 1-2）}$$

而信息增益即为两者的差值：

$$gain(A)=info(D)-infoA(D) \quad \text{（公式 1-3）}$$

ID3 算法就是在每次需要分裂时，计算每个属性的增益率，然后选择增益率最大的属性进行分裂。

对于特征属性为连续值，可以如此使用 ID3 算法：先将 D 中元素按照特征属性排序，则每两个相邻元素的中间点可以看做潜在分裂点，从第一个潜在分裂点开始，分裂 D 并计算两个集合的期望信息，具有最小期望信息的点称为这个属性的最佳分裂点，其信息期望作为此属性的信息期望。

五、实验目的：

编程实现决策树算法 ID3；理解算法原理。

六、实验内容：

利用 traindata.txt 的数据（75*5，第 5 列为标签）进行训练，构造决策树；利用构造好的决策树对 testdata.txt 的数据进行分类，并输出分类准确率。

要求：

（1）提交源代码及可执行文件。

（2）提交实验报告，内容包括：对代码的简单说明、运行结果的截图及说明等。

（3）需画出决策树，指明每个分支所对应的特征/属性，以及分裂值。

注：如用到了剪枝、限定深度等技巧（加分项），请加以说明。

七、实验器材（设备、元器件）：

PC 微机一台

八、实验步骤：

1.对数据集的预处理

在进行决策树构建之前，首先我们要针对训练集和测试集作相应的预处理，以加快训练速度和提升训练效果。在本实验中，数据集中每个特征的值是离散的

float 型数据，因此如果要采用决策树，必须对每个特征的值作**区间划分**，最终根据区间分为较少的类别来表示，否则每个特征的一个值将对应一个类别，这不仅会增加训练的开销，使决策树变得异常复杂，也会大大降低准确度。

因此，我对训练集中的每一个特征根据其类别做出箱线图（如图 1），以分析每个特征的区间分布。

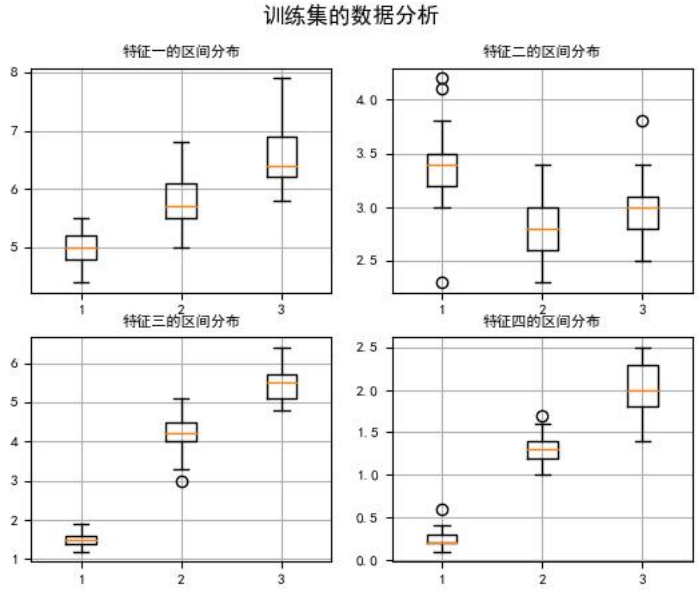

图 1 数据预处理分析

根据该箱线图，我对每个特征进行了区间划分（如表 1），最终**特征一划分为 4 类，特征二划分为 4 类，特征三划分为 3 类，特征四划分为 3 类**。

表 1 特征的区间划分

特征 / 类别	特征一	特征二	特征三	特征四
0	(0, 5.5]	(0, 2.5]	(0, 3]	(0, 1]
1	(5.5, 6]	(2.5, 3]	(3, 5]	(1, 1.5]
2	(6, 7]	(3, 3.5]	(5, +∞]	(1.5, +∞)
3	(7, +∞)	(3.5, +∞)	-	-

接着由该分类方法，可以对训练集以及测试集作批量的预处理，将每行每个特征值划分为所属的类别，最终结果如图 2 所示，左边为未经预处理的原始数据 traindata.txt，右边为预处理后的数据 transdata.txt。

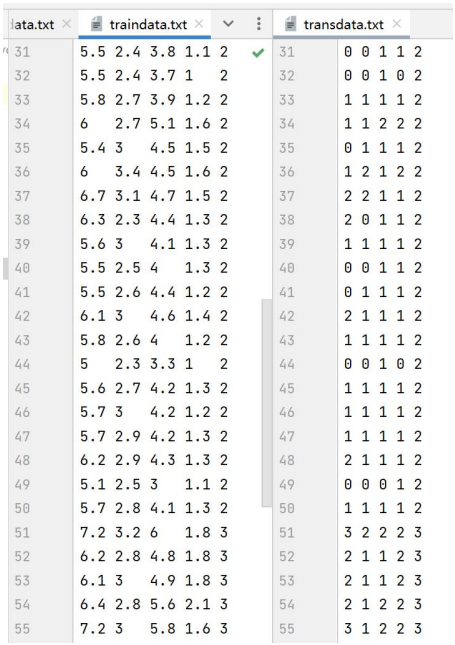

图 2 对数据集作预处理

2.决策树的预剪枝

决策树算法生成的决策树非常庞大，每个变量都被详细地考虑过。在每一个叶节点上，只要继续分支会有信息增益的情况，不管信息增益有多大，都会进行分支操作。如果我们用这个决策树来对训练集进行分类的话，那么这颗树的表现非常好。但是在测试集上的表现就远没有在训练集上的表现好，这就是**过拟合问题**。

而决策树的剪枝就是删掉一些不必要的逻辑判断，并且将子集合并。这样确实会造成在训练集上子集不纯的现象，但是因为我们最终目标是模型在测试集上的效果，所以牺牲在训练集上的效果换取解决测试集的过拟合问题这样的做法也是值得的。

考虑到数据集不算庞大且特征维度较小，因此本实验采用了最简单的剪枝方法——**预剪枝方法**。即在生成决策树的同时进行剪枝。正常决策树的生成是只要有信息增益就要进行分支，而预剪枝就是设定一个**阈值**，只有在信息增益大于这个阈值的时候（也即是在分类后的信息混乱程度减小程度大于一定标准的时候）才进行分类。如果在信息增益过小的情况下，即使存在信息增益的现象，也不会对其进行分支。本实验中经过多次调试，**预剪枝的阈值我设为了 0.1**，即只有在信息增益大于 0.1 时才进行分类。

3.算法设计

决策树算法设计

Input：训练数据集 D，特征集 A，阈值 ε；

Output：决策树 T。

1. 若 D 中所有实例**属于同一类 Ck**，则 T 为**单节点树**，并将类 Ck 作为该节点的类标记，返回 T；
2. 若 **A=∅** ，则 T 为单节点树，并将 D 中**实例数最大的类 Ck 作为该节点的类标记**，返回 T；
3. 否则，计算 A 中的各特征对 D 的信息增益，选择**信息增益最大的特征** Ag；
4. 如果 **Ag 的信息增益小于阈值 ε**，则置 T 为单节点树，并将 D 中实例数最大的类 Ck 作为该节点的类标记，返回 T；
5. 否则，对 Ag 的每一个可能值 ai，依 Ag=ai 将 D 分割为若干非空子集 Di，将 Di 中**实例数最大的类**作为标记，构建子节点，由节点以及子节点构成 T，返回 T；
6. 对第 i 个子节点，以 Di 为训练集，以 A-{Ag}为特征集，**递归地调用** 1-5 步，得到子树 Ti，返回 Ti。

4.具体实现

本次实现使用 python 语言进行编写，构造了 **DecisionTree 类**，其中集成了决策树的构造、训练以及测试等全部内容，因此将对这个类进行重点说明，详细代码见附件。

（1）DecisionTree 类的构造器：

```
class DecisionTree():
    def __init__(self, train_filepath, test_filepath):
        self.range_fea = [[5.5, 6, 7, 10], [2.5, 3, 3.5, 10], [3, 5, 10], [1, 1.5, 10]]
        self.fea_class_num = []
        for fea_range in self.range_fea:
            self.fea_class_num.append(len(fea_range))
        self.train_data, self.train_label = self.loadData(train_filepath)
        self.test_data, self.test_label = self.loadData(test_filepath)
```

DecisionTree 类是用于表示决策树的类，它包含 **range_fea**（数据预处理的分类区间）、fea_class_num（用于存储每个特征的类别个数，后面构造决策树时需要使用）、**train_data**（存放预处理后的训练集的特征数据）、**train_label**（存放训练集的标签数据）、**test_data**（存放预处理后的测试集的特征数据）以及 **test_label**（存放测试集的标签数据）六个类成员变量。

（2）DecisionTree.loadData（载入数据集及数据预处理）

```
def loadData(self, fileName):
    '''
    加载文件
    :param fileName:要加载的文件路径
    :return: 数据集和标签集
    '''
    # 存放数据及标记
    dataArr = []
    labelArr = []
    with open(fileName, 'r', encoding='utf-8') as infile:
        for line in infile:
            data_line = line.strip("\n").split()  # 去除首尾换行符，并按空格划分
            feature = [float(data) for data in data_line[0:4]]
            for i in range(len(feature)):
                fea = feature[i]
                range_each = self.range_fea[i]
                for j in range(len(range_each)):
                    if fea <= float(range_each[j]):
                        feature[i] = j
                        break
            dataArr.append(feature)
            labelArr.append(int(data_line[4]))
    # 返回数据集和标记
    return dataArr, labelArr
```

这个方法实现载入 txt 文件中的数据集数据并将预处理后的数据整合成特征及标签。其中预处理过程将使用成员变量 **range_fea** 来进行**特征值的分类**。

（3）DecisionTree.calc_H_D（计算数据集 D 的经验熵）以及 DecisionTree.calcH_D_A（计算经验条件熵）：

```
def calc_H_D(self, trainLabelArr):
    '''
    计算数据集D的经验熵
    :param trainLabelArr:当前数据集的标签集
    :return: 经验熵
    '''
    H_D = 0
    trainLabelSet = set([label for label in trainLabelArr])
    for i in trainLabelSet:
        # 计算|Ck|/|D|
        p = trainLabelArr[trainLabelArr == i].size / trainLabelArr.size
        # 对经验熵的每一项累加求和
        H_D += -1 * p * np.log2(p)
    return H_D
```

本方法用于计算一个数据集的经验熵，直接利用公式 1-1 编写即可。

```
def calcH_D_A(self, trainDataArr_DevFeature, trainLabelArr):
    '''
    计算经验条件熵
    :param trainDataArr_DevFeature:切割后只有feature那列数据的数组
    :param trainLabelArr: 标签集数组
    :return: 经验条件熵
    '''
    H_D_A = 0
    trainDataSet = set([label for label in trainDataArr_DevFeature])

    # 对于每一个特征取值遍历计算条件经验熵的每一项
    for i in trainDataSet:
        # 计算H(D|A)
        # trainDataArr_DevFeature[trainDataArr_DevFeature == i].size / trainDataArr_DevFeature.size:|Di| / |D|
        # calc_H_D(trainLabelArr[trainDataArr_DevFeature == i]):H(Di)
        H_D_A += trainDataArr_DevFeature[trainDataArr_DevFeature == i].size / trainDataArr_DevFeature.size \
                 * self.calc_H_D(trainLabelArr[trainDataArr_DevFeature == i])
    # 返回得出的条件经验熵
    return H_D_A
```

本方法用于计算一个数据集的条件经验熵，直接利用公式 1-2 编写即可。

（4）DecisionTree.calcBestFeature（得到数据集 D 最大的信息增益）

```
def calcBestFeature(self, trainDataList, trainLabelList):
    '''
    计算信息增益最大的特征
    :param trainDataList: 当前数据集
    :param trainLabelList: 当前标签集
    :return: 信息增益最大的特征及最大信息增益值
    '''
    trainDataArr = np.array(trainDataList)
    trainLabelArr = np.array(trainLabelList)
    featureNum = trainDataArr.shape[1]

    maxG_D_A = -1
    maxFeature = -1

    # 1.计算数据集D的经验熵H(D)
    H_D = self.calc_H_D(trainLabelArr)
    for feature in range(featureNum):
        trainDataArr_DevideByFeature = np.array(trainDataArr[:, feature].flat)
        # 2.计算信息增益G(D|A)    G(D|A) = H(D) - H(D | A)
        G_D_A = H_D - self.calcH_D_A(trainDataArr_DevideByFeature, trainLabelArr)
        if G_D_A > maxG_D_A:
            maxG_D_A = G_D_A
            maxFeature = feature
    return maxFeature, maxG_D_A
```

本方法用于计算一个数据集最大的信息增益以及对应的特征，首先需要利用 DecisionTree.calc_H_D 方法计算出数据集的经验熵，接着使用 DecisionTree.calcH_D_A 方法计算出数据集的条件经验熵，根据公式 1-3，信息增益=经验熵-条件经验熵，由此选出最大的信息增益并返回对应的特征。

（5）DecisionTree.createTree（创建决策树）

```
def createTree(self, *dataSet):
    '''
    递归创建决策树
    :param dataSet:(trainDataList, trainLabelList) <<-- 元祖形式
    :return:新的子节点或该叶子节点的值
    '''
    Epsilon = 0.1
    trainDataList = dataSet[0][0]
    trainLabelList = dataSet[0][1]
    # 打印信息：开始一个子节点创建，打印当前特征向量数目及当前剩余样本数目
    print('创建一个子节点，此时数据集中特征维度为{}，剩余样本数量为{}。'.format(len(trainDataList[0]), len(trainLabelList)))

    classDict = set(trainLabelList)
    # 如果D中所有实例属于同一类Ck，则置T为单节点数，并将Ck作为该节点的类，返回T
    if len(classDict) == 1:
        return trainLabelList[0]

    # 如果A为空集，则置T为单节点数，并将D中实例数最大的类Ck作为该节点的类，返回T
    if len(trainDataList[0]) == 0:
    # 返回当前标签集中占数目最大的标签
        return self.majorClass(trainLabelList)

    # 否则，计算A中特征值的信息增益，选择信息增益最大的特征Ag
    Ag, EpsilonGet = self.calcBestFeature(trainDataList, trainLabelList)

    # 如果Ag的信息增益比小于阈值Epsilon，则置T为单节点树，并将D中实例数最大的类Ck作为该节点的类，返回T
    if EpsilonGet < Epsilon:
        return self.majorClass(trainLabelList)

    # 否则，对Ag的每一可能值ai，依Ag=ai将D分割为若干非空子集Di，将Di中实例数最大的类作为标记，构建子节点，由节点及其子节点构成树T，返回T
    treeDict = {Ag: {}}
    class_num = self.fea_class_num[Ag]
    class_set = set([row[Ag] for row in trainDataList])
    for num in range(class_num):
        if num in class_set:
            treeDict[Ag][num] = self.createTree(self.getSubDataArr(trainDataList, trainLabelList, Ag, num))

    return treeDict
```

创建决策树的过程采用递归创建方式，递归返回情况有四种：（1）数据集中所有实例为同一类别，返回单节点树；（2）数据集中没有可用于继续分类的特征，返回单节点树；（3）最大的信息增益小于预剪枝阈值，返回单节点树；（4）递归结束，返回最终决策树。

（6）DecisionTree.predict（利用训练好的决策树模型进行预测）

```
def predict(self, testDataList, tree):
    '''
    预测标签
    :param testDataList:样本
    :param tree: 决策树
    :return: 预测结果
    '''
    # 死循环，直到找到一个有效地分类
    while True:
        (key, value), = tree.items()
        # 如果当前的value是字典，说明还需要遍历下去
        if type(tree[key]).__name__ == 'dict':
            # 获取目前所在节点的feature值，需要在样本中删除该feature
            dataVal = testDataList[key]
            del testDataList[key]
            # 将tree更新为其子节点的字典
            tree = value[dataVal]
            # 如果当前节点的子节点的值是int，就直接返回该int值
            if type(tree).__name__ == 'int':
                # 返回该节点值，也就是分类值
                return tree
        else:
            # 如果当前value不是字典，那就返回分类值
            return value
```

该方法使用前面创建的决策树来预测测试集数据，通过不断取出对应特征的

类别与决策树进行决策，直到最终取到字典中的 int 型变量，代表着已经实现分类。

九、实验数据及结果分析：

（1）决策树展示

创建决策树过程输出：

```
开始创建决策树...
创建一个子节点，此时数据集中特征维度为4，剩余样本数量为75。
创建一个子节点，此时数据集中特征维度为3，剩余样本数量为26。
创建一个子节点，此时数据集中特征维度为2，剩余样本数量为25。
创建一个子节点，此时数据集中特征维度为2，剩余样本数量为1。
创建一个子节点，此时数据集中特征维度为3，剩余样本数量为27。
创建一个子节点，此时数据集中特征维度为2，剩余样本数量为3。
创建一个子节点，此时数据集中特征维度为2，剩余样本数量为18。
创建一个子节点，此时数据集中特征维度为2，剩余样本数量为6。
创建一个子节点，此时数据集中特征维度为1，剩余样本数量为1。
创建一个子节点，此时数据集中特征维度为1，剩余样本数量为4。
创建一个子节点，此时数据集中特征维度为0，剩余样本数量为1。
创建一个子节点，此时数据集中特征维度为0，剩余样本数量为3。
创建一个子节点，此时数据集中特征维度为1，剩余样本数量为1。
创建一个子节点，此时数据集中特征维度为3，剩余样本数量为22。
创建一个子节点，此时数据集中特征维度为2，剩余样本数量为3。
创建一个子节点，此时数据集中特征维度为2，剩余样本数量为14。
创建一个子节点，此时数据集中特征维度为2，剩余样本数量为5。
决策树结构为： {2: {0: {2: {0: 1, 1: 2}}, 1: {2: {0: 2, 1: 2, 2: {1: {0: 3, 1: {0: {1: 3, 2: 3}}, 2: 2}}}}, 2: {0: {1: 3, 2: 3, 3: 3}}}}
```

可视化决策树如图 3 所示。

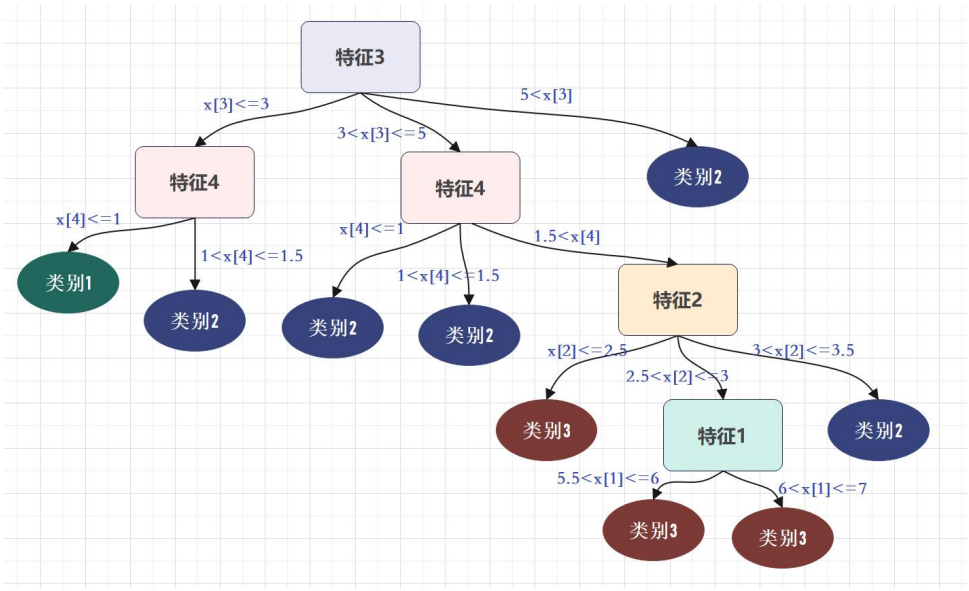

图 3 可视化决策树

（2）预测测试集

预测测试集输出为：

```
调用测试集测试中...
测试集样本0，标签为1，预测结果为1。
测试集样本1，标签为1，预测结果为1。
测试集样本2，标签为1，预测结果为1。
测试集样本3，标签为1，预测结果为1。
测试集样本4，标签为1，预测结果为1。
测试集样本5，标签为1，预测结果为1。
测试集样本6，标签为1，预测结果为1。
测试集样本7，标签为1，预测结果为1。
测试集样本8，标签为1，预测结果为1。
测试集样本9，标签为1，预测结果为1。
测试集样本10，标签为1，预测结果为1。
测试集样本11，标签为1，预测结果为1。
测试集样本12，标签为1，预测结果为1。
测试集样本13，标签为1，预测结果为1。
测试集样本14，标签为1，预测结果为1。
测试集样本15，标签为1，预测结果为1。
测试集样本16，标签为1，预测结果为1。
测试集样本17，标签为1，预测结果为1。
测试集样本18，标签为1，预测结果为1。
测试集样本19，标签为1，预测结果为1。
测试集样本20，标签为1，预测结果为1。
测试集样本21，标签为1，预测结果为1。
测试集样本22，标签为1，预测结果为1。
测试集样本23，标签为1，预测结果为1。
测试集样本24，标签为1，预测结果为1。
测试集样本25，标签为2，预测结果为2。
测试集样本26，标签为2，预测结果为2。
测试集样本27，标签为2，预测结果为2。
测试集样本28，标签为2，预测结果为2。
测试集样本29，标签为2，预测结果为2。
测试集样本30，标签为2，预测结果为2。
测试集样本31，标签为2，预测结果为2。
测试集样本32，标签为2，预测结果为2。
测试集样本33，标签为2，预测结果为2。
测试集样本34，标签为2，预测结果为2。
测试集样本35，标签为2，预测结果为2。
测试集样本36，标签为2，预测结果为2。
测试集样本37，标签为2，预测结果为2。
测试集样本38，标签为2，预测结果为2。
测试集样本39，标签为2，预测结果为2。
测试集样本40，标签为2，预测结果为2。
测试集样本41，标签为2，预测结果为2。
测试集样本42，标签为2，预测结果为2。
测试集样本43，标签为2，预测结果为2。
测试集样本44，标签为2，预测结果为2。
测试集样本45，标签为2，预测结果为2。
测试集样本46，标签为2，预测结果为2。
测试集样本47，标签为2，预测结果为2。
测试集样本48，标签为2，预测结果为2。
测试集样本49，标签为2，预测结果为2。
测试集样本50，标签为3，预测结果为3。
测试集样本51，标签为3，预测结果为3。
测试集样本52，标签为3，预测结果为3。
测试集样本53，标签为3，预测结果为3。
测试集样本54，标签为3，预测结果为3。
测试集样本55，标签为3，预测结果为3。
测试集样本56，标签为3，预测结果为3。
测试集样本57，标签为3，预测结果为3。
测试集样本58，标签为3，预测结果为3。
测试集样本59，标签为3，预测结果为3。
测试集样本60，标签为3，预测结果为3。
测试集样本61，标签为3，预测结果为3。
测试集样本62，标签为3，预测结果为3。
测试集样本63，标签为3，预测结果为3。
测试集样本64，标签为3，预测结果为3。
测试集样本65，标签为3，预测结果为3。
测试集样本66，标签为3，预测结果为3。
测试集样本67，标签为3，预测结果为3。
测试集样本68，标签为3，预测结果为3。
测试集样本69，标签为3，预测结果为2。
测试集样本70，标签为3，预测结果为3。
测试集样本71，标签为3，预测结果为3。
测试集样本72，标签为3，预测结果为3。
测试集样本73，标签为3，预测结果为3。
测试集样本74，标签为3，预测结果为3。
最终测试准确率为： 0.9866666666666667

总用时： 0.01599431037902832s

Process finished with exit code 0
```

在测试集中用该决策树进行预测时只在预测**第 69 个样本**时发生了 1 处错误，原始数据中该样本对应为第 3 类，而预测时将其预测为第 2 类。最终**分类准确率为 98.67%**，实验结果较为理想。

十、实验结论：

本次实验圆满完成，成功实现了所有实验要求，能正确地按规格输出结果。

十一、总结及心得体会：

本次实验我熟悉和掌握了决策树算法 ID3，理解了算法原理，并利用该算法成功构建出了一个决策树，最终准确率达到了 98%以上。使我对 ID3 算法有了更深入的理解和思考，对该算法的应用有了一定的了解。

十二、对本实验过程及方法、手段的改进建议：

在实际编写代码时，可以先根据算法设计写出伪代码，这样在实现时能更清晰且更不容易出错。

十三、代码附件

实验二、ID3 算法构建决策树

```
1.      import time
2.
3.      import numpy as np
4.
5.
6.      class DecisionTree():
7.          def __init__(self, train_filepath, test_filepath):
8.              self.range_fea = [[5.5, 6, 7, 10], [2.5, 3, 3.5, 10], [3, 5, 10], [1, 1.
    5, 10]]
9.              self.fea_class_num = []
10.             for fea_range in self.range_fea:
11.                 self.fea_class_num.append(len(fea_range))
12.             self.train_data, self.train_label = self.loadData(train_filepath)
13.             self.test_data, self.test_label = self.loadData(test_filepath)
14.
15.         def loadData(self, fileName):
16.             '''
17.             加载文件
18.             :param fileName:要加载的文件路径
19.             :return: 数据集和标签集
20.             '''
21.             # 存放数据及标记
22.             dataArr = []
23.             labelArr = []
24.             with open(fileName, 'r', encoding='utf-8') as infile:
25.                 for line in infile:
26.                     data_line = line.strip("\n").split()  # 去除首尾换行符，并按空格划分
```

```
                    feature = [float(data) for data in data_line[0:4]]
                    for i in range(len(feature)):
                        fea = feature[i]
                        range_each = self.range_fea[i]
                        for j in range(len(range_each)):
                            if fea <= float(range_each[j]):
                                feature[i] = j
                                break
                    dataArr.append(feature)
                    labelArr.append(int(data_line[4]))
            # 返回数据集和标记
            return dataArr, labelArr

        def majorClass(self, labelArr):
            '''
            找到当前标签集中占数目最大的标签
            :param labelArr: 标签集
            :return: 最大的标签
            '''
            # 建立字典，用于不同类别的标签技术
            classDict = {}
            # 遍历所有标签
            for i in range(len(labelArr)):
                if labelArr[i] in classDict.keys():
                    classDict[labelArr[i]] += 1
                else:
                    classDict[labelArr[i]] = 1
            # 对字典依据值进行降序排序
            classSort = sorted(classDict.items(), key=lambda x: x[1], reverse=True)
            # 返回最大一项的标签，即占数目最多的标签
            return classSort[0][0]

        def calc_H_D(self, trainLabelArr):
            '''
            计算数据集 D 的经验熵
            :param trainLabelArr:当前数据集的标签集
            :return: 经验熵
            '''
            H_D = 0
            trainLabelSet = set([label for label in trainLabelArr])
            for i in trainLabelSet:
                # 计算|Ck|/|D|
                p = trainLabelArr[trainLabelArr == i].size / trainLabelArr.size
                # 对经验熵的每一项累加求和
```

```
71.                 H_D += -1 * p * np.log2(p)
72.             return H_D
73.
74.         def calcH_D_A(self, trainDataArr_DevFeature, trainLabelArr):
75.             '''
76.             计算经验条件熵
77.             :param trainDataArr_DevFeature:切割后只有 feature 那列数据的数组
78.             :param trainLabelArr: 标签集数组
79.             :return: 经验条件熵
80.             '''
81.             H_D_A = 0
82.             trainDataSet = set([label for label in trainDataArr_DevFeature])
83.
84.             # 对于每一个特征取值遍历计算条件经验熵的每一项
85.             for i in trainDataSet:
86.                 # 计算H(D|A)
87.                 # trainDataArr_DevFeature[trainDataArr_DevFeature == i].size / train
    DataArr_DevFeature.size:|Di| / |D|
88.                 # calc_H_D(trainLabelArr[trainDataArr_DevFeature == i]):H(Di)
89.                 H_D_A += trainDataArr_DevFeature[trainDataArr_DevFeature == i].size
    / trainDataArr_DevFeature.size \
90.                          * self.calc_H_D(trainLabelArr[trainDataArr_DevFeature == i]
    )
91.             # 返回得出的条件经验熵
92.             return H_D_A
93.
94.         def calcBestFeature(self, trainDataList, trainLabelList):
95.             '''
96.             计算信息增益最大的特征
97.             :param trainDataList: 当前数据集
98.             :param trainLabelList: 当前标签集
99.             :return: 信息增益最大的特征及最大信息增益值
100.            '''
101.            trainDataArr = np.array(trainDataList)
102.            trainLabelArr = np.array(trainLabelList)
103.            featureNum = trainDataArr.shape[1]
104.
105.            maxG_D_A = -1
106.            maxFeature = -1
107.
108.            # 1.计算数据集D 的经验熵H(D)
109.            H_D = self.calc_H_D(trainLabelArr)
110.            for feature in range(featureNum):
111.                trainDataArr_DevideByFeature = np.array(trainDataArr[:, feature].fla
```

```
    t)
112.                 # 2.计算信息增益G(D|A)    G(D|A) = H(D) - H(D | A)
113.                 G_D_A = H_D - self.calcH_D_A(trainDataArr_DevideByFeature, trainLabe
    lArr)
114.                 if G_D_A > maxG_D_A:
115.                     maxG_D_A = G_D_A
116.                     maxFeature = feature
117.             return maxFeature, maxG_D_A
118.
119.         def getSubDataArr(self, trainDataArr, trainLabelArr, A, a):
120.             '''
121.             更新数据集和标签集
122.             :param trainDataArr:要更新的数据集
123.             :param trainLabelArr: 要更新的标签集
124.             :param A: 要去除的特征索引
125.             :param a: 当 data[A]== a 时，说明该行样本时要保留的
126.             :return: 新的数据集和标签集
127.             '''
128.             retDataArr = []
129.             retLabelArr = []
130.             # 对当前数据的每一个样本进行遍历
131.             for i in range(len(trainDataArr)):
132.                 # 如果当前样本的特征为指定特征值 a
133.                 if trainDataArr[i][A] == a:
134.                     # 那么将该样本的第 A 个特征切割掉，放入返回的数据集中
135.                     retDataArr.append(trainDataArr[i][0:A] + trainDataArr[i][A + 1:]
    )
136.                     retLabelArr.append(trainLabelArr[i])
137.             return retDataArr, retLabelArr
138.
139.         def createTree(self, *dataSet):
140.             '''
141.             递归创建决策树
142.             :param dataSet:(trainDataList， trainLabelList) <<-- 元祖形式
143.             :return:新的子节点或该叶子节点的值
144.             '''
145.             Epsilon = 0.1
146.             trainDataList = dataSet[0][0]
147.             trainLabelList = dataSet[0][1]
148.             # 打印信息：开始一个子节点创建，打印当前特征向量数目及当前剩余样本数目
149.             print('创建一个子节点，此时数据集中特征维度为{}，剩余样本数量为{}。
    '.format(len(trainDataList[0]), len(trainLabelList)))
150.
151.             classDict = set(trainLabelList)
```

```
152.             # 如果D中所有实例属于同一类Ck，则置T为单节点数，并将Ck作为该节点的类，返回T
153.             if len(classDict) == 1:
154.                 return trainLabelList[0]
155.
156.             # 如果A为空集，则置T为单节点数，并将D中实例数最大的类Ck作为该节点的类，返回
    T
157.             if len(trainDataList[0]) == 0:
158.                 # 返回当前标签集中占数目最大的标签
159.                 return self.majorClass(trainLabelList)
160.
161.             # 否则，计算A中特征值的信息增益，选择信息增益最大的特征Ag
162.             Ag, EpsilonGet = self.calcBestFeature(trainDataList, trainLabelList)
163.
164.             # 如果Ag的信息增益比小于阈值Epsilon，则置T为单节点树，并将D中实例数最大的类
    Ck作为该节点的类，返回T
165.             if EpsilonGet < Epsilon:
166.                 return self.majorClass(trainLabelList)
167.
168.             # 否则，对Ag的每一可能值ai，依Ag=ai将D分割为若干非空子集Di，将Di中实例数最
    大的类作为标记，构建子节点，由节点及其子节点构成树T，返回T
169.             treeDict = {Ag: {}}
170.             class_num = self.fea_class_num[Ag]
171.             class_set = set([row[Ag] for row in trainDataList])
172.             for num in range(class_num):
173.                 if num in class_set:
174.                     treeDict[Ag][num] = self.createTree(self.getSubDataArr(trainData
    List, trainLabelList, Ag, num))
175.
176.             return treeDict
177.
178.         def predict(self, testDataList, tree):
179.             '''
180.             预测标签
181.             :param testDataList:样本
182.             :param tree: 决策树
183.             :return: 预测结果
184.             '''
185.             # 死循环，直到找到一个有效地分类
186.             while True:
187.                 (key, value), = tree.items()
188.                 # 如果当前的value是字典，说明还需要遍历下去
189.                 if type(tree[key]).__name__ == 'dict':
190.                     # 获取目前所在节点的feature值，需要在样本中删除该feature
191.                     dataVal = testDataList[key]
```

```
                    del testDataList[key]
                    # 将tree 更新为其子节点的字典
                    tree = value[dataVal]
                    # 如果当前节点的子节点的值是int，就直接返回该int 值
                    if type(tree).__name__ == 'int':
                        return tree
                else:
                    return value

        def model_test(self, testDataList, testLabelList, tree):
            '''
            测试准确率
            :param testDataList:待测试数据集
            :param testLabelList: 待测试标签集
            :param tree: 训练集生成的树
            :return: 准确率
            '''
            # 错误次数计数
            errorCnt = 0
            for i in range(len(testDataList)):
                # 判断预测与标签中结果是否一致
                pre_label = self.predict(testDataList[i], tree)
                print("测试集样本{}，标签为{}，预测结果为{}。
    ".format(i, testLabelList[i], pre_label))
                if testLabelList[i] != pre_label:
                    errorCnt += 1
            return 1 - errorCnt / len(testDataList)


    if __name__ == '__main__':
        # 开始时间
        start = time.time()

        # 创建决策树
        print('开始创建决策树...')
        model = DecisionTree("traindata.txt", "testdata.txt")
        tree = model.createTree((model.train_data, model.train_label))
        print('决策树结构为：', tree)
        print()

        # 测试准确率
        print('调用测试集测试中...')
        accur = model.model_test(model.test_data, model.test_label, tree)
        print('最终测试准确率为：', accur)
```

```
        print()

        # 结束时间
        end = time.time()
        print('总用时: {}s'.format(end - start))
```

绘画特征的箱线图

```
import matplotlib.pyplot as plt

filepath  = "traindata.txt"
with open(filepath, 'r', encoding='utf-8') as infile:
    # 每行分开读取
    data1_1 = []
    data2_1 = []
    data3_1 = []
    data4_1 = []

    data1_2 = []
    data2_2 = []
    data3_2 = []
    data4_2 = []

    data1_3 = []
    data2_3 = []
    data3_3 = []
    data4_3 = []
    for line in infile:
        data_line = line.strip("\n").split()  # 去除首尾换行符，并按空格划分
        if int(data_line[4]) == 1:
            data1_1.append(float(data_line[0]))
            data2_1.append(float(data_line[1]))
            data3_1.append(float(data_line[2]))
            data4_1.append(float(data_line[3]))
        elif int(data_line[4]) == 2:
            data1_2.append(float(data_line[0]))
            data2_2.append(float(data_line[1]))
            data3_2.append(float(data_line[2]))
            data4_2.append(float(data_line[3]))
        elif int(data_line[4]) == 3:
            data1_3.append(float(data_line[0]))
            data2_3.append(float(data_line[1]))
```

```
                data3_3.append(float(data_line[2]))
                data4_3.append(float(data_line[3]))
    labels = ["1","2","3"]

    plt.figure()
    plt.rcParams['font.sans-serif']=['SimHei']
    plt.rcParams['axes.unicode_minus'] = False
    plt.subplot(221)
    plt.grid(True)
    plt.xticks(fontsize=8)
    plt.yticks(fontsize=8)
    plt.boxplot([data1_1,data1_2,data1_3],labels=labels)
    plt.title("特征一的区间分布",fontsize=8)

    plt.subplot(222)
    plt.grid(True)
    plt.xticks(fontsize=8)
    plt.yticks(fontsize=8)
    plt.boxplot([data2_1,data2_2,data2_3],labels=labels)
    plt.title("特征二的区间分布",fontsize=8)

    plt.subplot(223)
    plt.grid(True)
    plt.xticks(fontsize=8)
    plt.yticks(fontsize=8)
    plt.boxplot([data3_1,data3_2,data3_3],labels=labels)
    plt.title("特征三的区间分布",fontsize=8)

    plt.subplot(224)
    plt.grid(True)
    plt.xticks(fontsize=8)
    plt.yticks(fontsize=8)
    plt.boxplot([data4_1,data4_2,data4_3],labels=labels)
    plt.title("特征四的区间分布",fontsize=8)

    plt.suptitle("训练集的数据分析")
    plt.show()
```

报告评分：

指导教师签字：