

一、实验室名称：主楼 A2-412

二、实验项目名称：认识数据与数据预处理

三、实验学时：3

四、实验原理

1、数据属性最小最大归一化（可尝试其它方法：如 ZSCORE）

$$v' = \frac{v - \min_A}{\max_A - \min_A} (new_max_A - new_min_A) + new_min_A$$

其中 v 是属性 A 的某个观测值， \min_A 和 \max_A 分别是属性 A 的最小值和最大值

上述公式将 A 属性的取值映射到区间[new_minA,new_maxA]

如果令 $new_maxA = 1$, $new_minA = 0$, 则将 A 属性映射到区间[0,1], 实现了数据归一化

2、缺失值处理

对于数据中属性的缺失值，使用该属性的平均值来填补缺失值

例如：

表 4.2.1 示例数据

V1	V2	V3	V4
0.2	0.4	0.3	0.4
0.8	0.3	0.4	0.4
0.1	0.0	0.3	0.6
?	0.4	0.3	0.5
0.4	0.6	0.3	0.4
0.2	0.4	0.3	0.5
0.1	0.4	0.3	0.5

Strategy:

$$\text{Mean} = (0.2+0.8+0.1+0.4+0.2+0.1)/6 = 0.3$$

$$? = 0.3$$

3、特征筛选

信息增益是用来进行特征筛选的常用算法，基本思想是选择那些特征对分类变量 Y 信息增

益大，删除那些对分类无用的特征。

信息熵：

$$H(X) = - \sum_{i=1}^n p(x_i) \log p(x_i).$$

条件信息熵：

$$\begin{aligned} H(Y|X) &\equiv \sum_{x \in \mathcal{X}} p(x) H(Y|X=x) \\ &= - \sum_{x \in \mathcal{X}} p(x) \sum_{y \in \mathcal{Y}} p(y|x) \log p(y|x) \end{aligned}$$

信息增益：

$$IG(Y|X) = H(Y) - H(Y|X)$$

五、实验目的

- 1、了解 Weka 工具包及 Eclipse 编程平台
- 2、认识和了解数据
- 3、对数据能进行简单的预处理

六、实验内容：

- 1、安装并配置 Java、Eclipse 和 Weka（已配置）
- 2、使用图形界面的 Weka 工具包，完成数据归一化、缺失值处理、特征筛选的数据预处理操作
- 3、在 Eclipse 下调用 Weka.jar 包，完成数据归一化、缺失值处理、特征筛选的数据预处理操作

七、实验器材（设备、元器件）：台式机、笔记本（MagicBook）

八、实验步骤

- 1、配置实验环境
- 2.1、使用 Weka 图形界面工具完成数据归一化
 - 1) 打开 Weka GUI 工具
 - 2) 选择 Explorer
 - 3) 点击 Open file
 - 4) 选择数据文件 iris.arff（在目录..../weka/data 下）
 - 5) 在 Filter 栏目下，选择 Choose

- 6) 选择 weka->filters->unsupervised->attribute->Normalize
- 7) 点击 “Normalize - S 1.0 - T 0.0” 所在区域可以调整参数，由于本例选择将数据归一化到[0,1]，使用默认的参数即可实现，点击 Apply 将数据归一化
- 8) 查看数据各个属性的值，已经归一化到[0,1]（注：其中 class 不是数据属性，而是数据类别标签，不会被归一化），将实验结果截图放入实验报告中

2.2、使用 Weka 图形界面工具完成数据缺失值处理

过程略，提示：数据使用 labor.arff，filter 使用 ReplaceMissingValues，参数默认

2.3、使用 Weka 图形界面工具完成特征筛选

过程略，提示：数据使用 iris.arff，filter 使用 AttributeSelection，其中参数 evaluator 选择 InfoGainAttributeEval，search 使用 Ranker，需要调节 Ranker 的参数，选择最大特征数目（需小于原始的特征数目）

3.1、在 Eclipse 下调用 Weka.jar 包完成数据归一化

- 1) 首先，新建一个 Java 工程
- 2) 接下来，导入 weka.jar
- 3) 创建一个 package 包：New -> package -> 输入 cn.uestc.preprocessing
- 4) 创建 Java 文件：在包下面创建一个新的 class: TestNormalize.java New -> Class -> 输入 TestNormalize
- 5) 编辑 TestNormalize.java 文件，导入包：

```
import weka.core.Attribute;
import weka.core.Instance;
import weka.core Instances;
import weka.core.converters.ConverterUtils.DataSource;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.Normalize;
```

- 6) 在 main 函数中完成数据归一化，相关代码如下：

```
//读取数据
DataSource source = new DataSource("D:/weka-3-7-7/weka-3-7-7/data/iris.arff"); //获取数据源
Instances instances = source.getDataSet(); //导入数据
```

//归一化

```
System.out.println("Step 2. 归一化...");
Normalize norm = new Normalize(); //建立一个归一化 filter
norm.setInputFormat(instances); //为 filter 导入数据
Instances newInstances = Filter.useFilter(instances, norm); //得到归一化后的数据
//打印结果 (printAttribute 函数在后面给出)
printAttribute(newInstances);
```

- 7) 观察数据是否已经归一化，打印实验结果，截图

printAttribute 函数，用于输出属性的值，然后截图

```
public static void printAttribute(Instances instances)
{
    int numAttributes = instances.numAttributes();
    for(int i = 0; i < numAttributes ;++i)
    {
        Attribute attribute = instances.attribute(i);
    }
}
```

```

        System.out.print(attribute.name() + "      ");
    }
    System.out.println();
    //打印实例
    int numofInstance = instances.numInstances();
    for(int i = 0; i < numofInstance; ++i)
    {
        Instance instance = instances.instance(i);
        System.out.print(instance.toString() + "      " + '\n' );
    }
}

```

3.2、在 Eclipse 下调用 Weka.jar 包完成数据缺失值处理

过程略，在处理缺失值时，可以自己编写函数完成数据缺失值处理，也可以调用 weka 包中的 ReplaceMissingValues 类来处理缺失值，操作过程与归一化处理类似，示例代码：

```

ReplaceMissingValues rmv = new ReplaceMissingValues();
rmv.setInputFormat(instances);
Instances newInstances = Filter.useFilter(instances, rmv);

```

3.3、在 Eclipse 下调用 Weka.jar 包完成特征筛选

过程略，提示：建立一个 AttributeSelection 对象，该对象需设置参数 Evaluator 和 Search，分别选择是 InfoGainAttributeEval 对象和 Ranker 对象。其中 InfoGainAttributeEval 对象用来评估特征；Ranker 对象负责根据对属性的评估筛选属性，需要设置特征阈值和筛选特征的数目。示例代码：

```

InfoGainAttributeEval ae = new InfoGainAttributeEval();//选择 evaluator 为信息增益
Ranker ranker = new Ranker();//评估函数选择 ranker
ranker.setNumToSelect(3);//设置筛选的最大特征数目
ranker.setThreshold(0.0);//评估特征值低于该阈值的特征被筛去，在此表示只留下信息增益大于 0 的特征
AttributeSelection as = new AttributeSelection();//建立特征筛选对象
as.setEvaluator(ae);//设置筛选对象所用的评估函数
as.setSearch(ranker);//设置筛选对象的选择函数
as.setInputFormat(instances);//为该特征筛选对象传入数据源

```

九、实验数据及结果分析

9.1 使用 Weka 图形界面工具完成数据归一化

使用 Weka 图形界面工具：

打开 Weka GUI 工具，选择 Explorer，点击 Open file，选择数据文件 iris.arff，在 Filter 栏目下选择 Choose，选择 weka->filters->unsupervised->attribute->Normalize，点击“Normalize - S 1.0 - T 0.0”。

查看数据各个属性的值，已经归一化到[0,1]，以下为实验结果截图：



图 9.1.1 Weka 归一化实验

以上截图显示了在 Weka 中对归一化后数据的描述，可见除了“class”类数据，其他数据的取值范围均为 0 到 1，可见成功完成归一化。

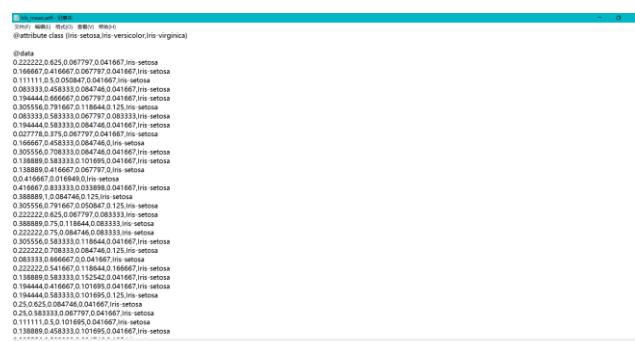


图 9.1.2 Weka 归一化实验

该截图展示了归一化后的数据。

使用 Python:

首先为数据读入函数：

```
def loadIris(address):#加载数据
    spf=pd.read_csv(address,sep=',',index_col=False,header=None)
    print(spf)
    strs=spf[4]
    spf.drop([0,4],axis=1,inplace=True)
    print(spf)
    return spf.values,strs
```

图 9.1.3 loadIris 函数

然后是核心函数，归一化函数：

```
def normalization(data_matrix):#归一化
    e=1e-5
    for c in range(4):
        maxNum=np.max(data_matrix[:,c])
        minNum=np.min(data_matrix[:,c])
        data_matrix[:,c]=(data_matrix[:,c]-minNum+e)/(maxNum-minNum+e)
    return data_matrix
```

图 9.1.4 normalization 函数

最后是程序入口：

```
if __name__=='__main__':#主函数
    filepath='G:\\大学\\数据挖掘\\实验1\\iris.txt'
    writepath='G:\\大学\\数据挖掘\\实验1\\iris_normal.txt'
    data_matrix,str_name=loadIris(filepath)
    data_matrix=normalization(data_matrix)
    spf=pd.DataFrame(data_matrix)
    strs=str_name.values
    spf.insert(4,strs)
    #spf_to_csv(writepath,index=False,header=False)#将归一化的数据写入txt
```

图 9.1.5 程序入口

使用 Python 程序完成归一化处理，实验结果截图如下，源码见附录。

图 9.1.6 Python 归一化实验（处理后数据）

可见两种方法得到的结果大体相同，唯一有区别的地方在于两种方法的结果小数位数不同，但这一区别完全可以通过设置标准输出格式来解决。

9.2 缺失值处理

使用 Weka 图形界面工具：

数据使用 labor.arff，filter 使用 ReplaceMissingValues，参数默认。实验结果截图如下：

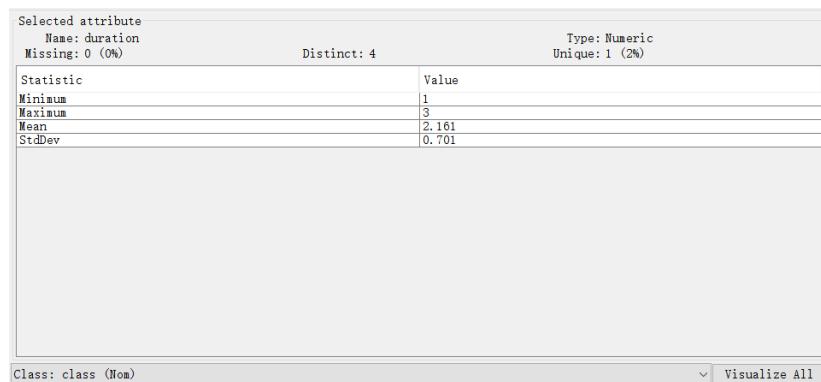


图 9.2.1 Weka 缺失值处理实验

```

@data
1,3,971739,3,913333,none,40,empl,contr,7,444444,2,no,11,average,yes,half,yes,full,good
2,4,5,5,8,3,913333,none,35,ret,allw,7,444444,4,8,70968,yes,11,below,average,yes,full,yes,full,good
2,1,60714,3,90571,3,971739,3,913333,8,empl,contr,7,444444,5,no,11,generous,yes,half,yes,full,good
3,3,7,4,5,tc,36,039216,empl,contr,7,444444,4,87,0968,yes,11,09434,below,average,yes,half,yes,full,good
3,4,5,4,5,5,none,40,empl,contr,7,444444,4,87,0968,no,12,average,yes,half,yes,full,good
2,2,3,9,4,3,913333,none,36,empl,contr,7,444444,4,87,0968,no,12,average,yes,half,yes,full,good
3,4,5,5,tc,38,039216,empl,contr,7,444444,4,87,0968,no,12,generous,yes,half,yes,full,good
3,6,9,4,8,2,3,none,40,empl,contr,7,444444,3,no,12,below,average,yes,half,yes,full,good
2,3,7,9,13333,none,36,empl,contr,12,25,3,yes,11,below,average,yes,half,yes,full,good
1,5,7,3,971739,3,913333,none,40,empl,contr,7,444444,4,no,11,generous,yes,half,yes,full,good
3,5,4,4,6,none,36,empl,contr,7,444444,4,87,0968,yes,11,09434,below,average,yes,half,yes,full,good
2,6,4,6,4,3,913333,none,36,empl,contr,7,444444,4,87,0968,no,12,average,yes,half,yes,full,good
2,3,5,4,3,913333,none,40,empl,contr,7,444444,4,87,0968,no,10,below,average,yes,half,yes,full,bad
3,3,5,4,5,1,tc,37,empl,contr,7,444444,4,87,0968,no,10,below,average,yes,half,yes,full,good
1,3,971739,3,913333,none,36,empl,contr,7,444444,10,no,11,generous,yes,half,yes,full,good
2,4,5,4,3,913333,none,37,empl,contr,7,444444,4,87,0968,no,11,average,yes,full,yes,full,good
1,2,8,3,971739,3,913333,none,35,empl,contr,7,444444,2,no,12,below,average,yes,half,yes,full,good
1,2,1,3,971739,3,913333,tc,40,ret,allw,2,3,no,9,below,average,yes,half,yes,none,bad
1,2,3,971739,3,913333,none,38,empl,contr,7,444444,4,87,0968,yes,11,average,yes,none,none,bad
2,4,5,4,3,913333,none,38,empl,contr,7,444444,4,87,0968,yes,11,average,yes,full,yes,full,good
2,4,3,4,4,3,913333,none,38,empl,contr,7,444444,4,no,12,generous,yes,half,yes,full,good
2,2,5,3,3,913333,none,40,none,7,444444,4,87,0968,no,11,09434,below,average,yes,half,yes,full,good
3,3,5,4,4,6,tc,27,empl,contr,7,444444,4,87,0968,yes,11,09434,below,average,yes,half,yes,full,good
2,4,5,4,3,913333,none,40,empl,contr,7,444444,4,no,10,generous,yes,half,yes,full,good
1,6,3,971739,3,913333,none,38,empl,contr,8,3,no,9,generous,yes,half,yes,full,good
3,2,2,2,none,40,none,7,444444,4,87,0968,no,10,below,average,yes,half,yes,full,good
2,4,5,4,3,913333,none,38,empl,contr,7,444444,4,87,0968,yes,11,average,yes,full,yes,none,yes,half,good
2,3,3,3,913333,none,33,empl,contr,7,444444,4,87,0968,yes,12,generous,yes,half,yes,full,good
2,5,4,3,913333,none,37,empl,contr,7,444444,5,no,11,below,average,yes,full,yes,full,good
2,3,2,5,3,913333,none,35,none,7,444444,4,87,0968,no,10,average,yes,half,yes,full,bad
3,4,5,4,5,5,none,40,empl,contr,7,444444,4,87,0968,no,11,average,yes,half,yes,full,good

```

图 9.2.2 Weka 缺失值处理实验（处理后数据）

第一张截图显示 Missing 为 0%，即缺失值全部被填充；第二张截图也没有发现缺失值，

说明操作成功。

使用 Python:

写代码如下：

首先为数据读入函数：

```

def LoadLabor(address): # 加载数据
    spf = pd.read_csv(address, sep=',', index_col=False, header=None)
    column = ['duration', 'wage-increase-first-year', 'wage-increase-second-year', 'wage-increase-third-year',
              'cost-of-living-adjustment', 'working-hours', 'pension', 'standby-pay',
              'shift-differential', 'education-allowance', 'statutory-holidays', 'vacation',
              'longterm-disability-assistance', 'contribution-to-dental-plan', 'bereavement-assistance',
              'contribution-to-health-plan', 'class'] # 所有标签
    spf.columns = column
    str_typeName = ['cost-of-living-adjustment', 'pension', 'education-allowance',
                    'vacation', 'longterm-disability-assistance', 'contribution-to-dental-plan',
                    'bereavement-assistance', 'contribution-to-health-plan', 'class'] # 字符数据标签

    str2numeric = {}
    str2numeric['?'] = '-1'
    return spf, str2numeric, str_typeName

```

图 9.2.3 loadLabor 函数

将数据加载完毕后使用缺失值填充函数，如下：

```

def fillMissData(spf, str_typeName): # 缺失值填充
    row, col = spf.shape
    columns = spf.columns
    for column_name in columns: # 将'?'替换为'-1'
        if column_name not in str_typeName:
            tmp = spf[column_name]
            for i in range(len(tmp)):
                if tmp[i] != '?':
                    tmp[i] = float(tmp[i])
            ave = np.average(tmp[tmp != '?'])
            tmp[tmp == '?'] = ave
            spf[column_name] = tmp
        else:
            v = spf[column_name].values
            v1 = v[v != '-1']
            c = Counter(v1)
            cc = c.most_common(1)
            v[v == '-1'] = cc[0][0]

```

图 9.2.4 fillMissDATa 函数

最后使用主函数完成一次缺失值填充实验：

```

if __name__ == '__main__': # 主函数
    filepath = 'G:\\大学\\数据挖掘\\laborMissing.txt'
    fillfilepath = 'G:\\大学\\数据挖掘\\laborMissing_handle.txt'
    spf, str2numeric, str_typeName = loadLabor(filepath)
    spf = fillMissData(spf, str_typeName)
    spf.to_csv(fillfilepath, index=False, header=False)

```

图 9.2.5 程序入口

使用 Python 程序完成缺失值处理，实验结果截图如下，源码见附录。

```

laborMissing_Handle.py:1:1
if __name__ == '__main__':
    filepath = 'G:\\大学\\数据挖掘\\laborMissing.txt'
    fillfilepath = 'G:\\大学\\数据挖掘\\laborMissing_handle.txt'
    spf, str2numeric, str_typeName = loadLabor(filepath)
    spf = fillMissData(spf, str_typeName)
    spf.to_csv(fillfilepath, index=False, header=False)

```

图 9.2.6 Python 缺失值处理实验

利用 Python 进行归一化操作后打开生成的 txt 文件（如上），发现原来“？”处的数据

都被填充，操作成功。

9.3 特征筛选

使用 Weka 图形界面工具：

数据使用 iris.arff, 数据有包括 sepalwidth 在内的 5 个特征, filter 使用 AttributeSelection, 其中参数 evaluator 选择 InfoGainAttributeEval, search 使用 Ranker, 调节 Ranker 的参数, 选择最大特征数目为 4。实验结果截图如下：

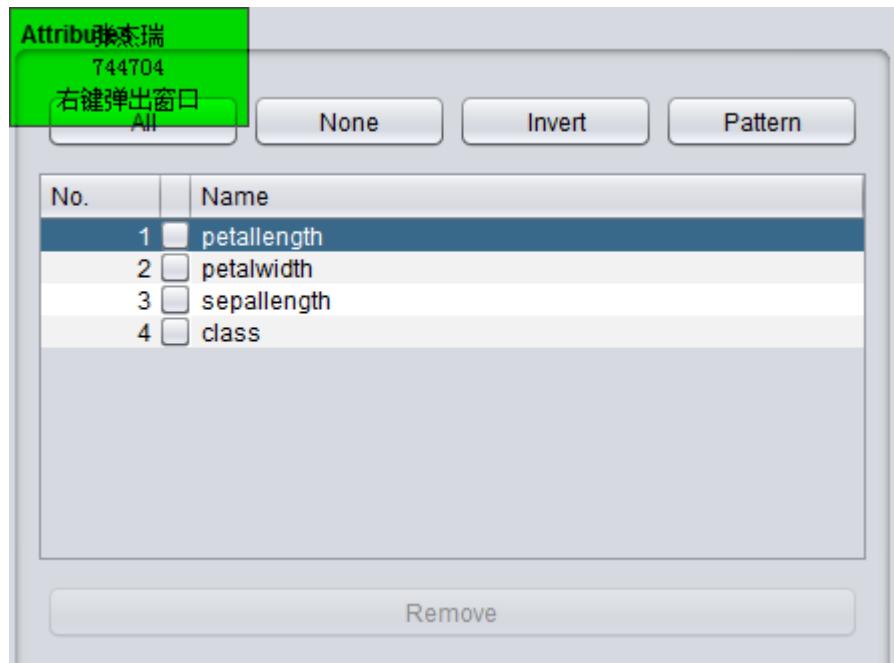


图 9.3.1 Weka 特征筛选实验

可见我们的确得到了 4 个主要特征，特征 sepalwidth 被剔除，完成了特征筛选功能。

使用 Python:

首先是数据加载函数：

```
def loadIris(address):#加载数据
    spf=pd.read_csv(address,sep=',',index_col=False,header=None)
    strs=spf[4]
    spf.drop([4],axis=1,inplace=True) #此处去除了class特征
    return spf.values,strs
```

图 9.3.2 loadIris 函数

然后是计算信息增益的函数，如下：

```

def getEnergy(c,data,label):#计算增益
    dataLen=len(label)
    energy=0.0
    for key,value in c.items():
        c[key]/=float(dataLen)
        label_picked=label[data==key]
        l=Counter(label_picked)
        e=0.0
        for k,v in l.items():
            r=v/float(value)
            e-=r*np.log2(r)
            energy+=c[key]*e
    return energy

```

图 9.3.3 getEnergy 函数

然后是特征筛选函数，选出增益最高的标签，并返回：

```

def featureSelection(features,label):#根据增益进行特征筛选
    featureLen=len(features[0,:])
    label_count=Counter(label)
    samples_energy=0.0
    data_len=len(label)
    for i in label_count.keys():
        label_count[i]/=float(data_len)
        samples_energy+=label_count[i]*np.log2(label_count[i])
    informationGain=[]
    for f in range(featureLen):
        af=features[:,f]
        minf=np.min(af)
        maxf=np.max(af)+1e-4
        width=(maxf-minf)/10.0
        d=(af-minf)/width
        dd=np.floor(d)
        c=Counter(dd)
        sub_energy=getEnergy(c,dd,label)

        informationGain.append(samples_energy-sub_energy)

```

图 9.3.4 featureSelection 函数

最后是程序入口：

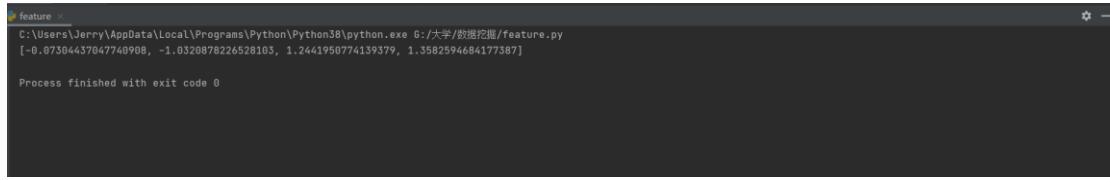
```

if __name__=='__main__':#主函数
    filepath='iris.txt'
    data_matrix,str_name=loadIris(filepath)
    informationGain=featureSelection(data_matrix,str_name.values)
    print(informationGain)

```

图 9.3.5 程序入口

编程，通过引入 Python 库、读取文件、计算特征的信息增益等步骤完成程序设计，实验结果截图如下：



```
feature <input>
C:\Users\Jerry\AppData\Local\Programs\Python\Python38\python.exe G:/大学\数据挖掘/feature.py
[-0.07304437047740908, -1.0320878226528103, 1.2441950774139379, 1.3582594684177387]

Process finished with exit code 0
```

图 9.3.6 Python 特征筛选实验

[-0.07304437047740908,-1.0320878226528103,1.2441950774139379,1.358259468417738]

为输出结果，说明第 4 个特征具有最高的信息增益，第 2 个特征最低。若要选择特征，应按照 4, 3, 1, 2 的顺序选择，能得到最优结果。

9.4 数据可视化

以生成的 laborMissing_handle.txt 文件为例展示数据可视化，使用 matlab 的 hist 函数。

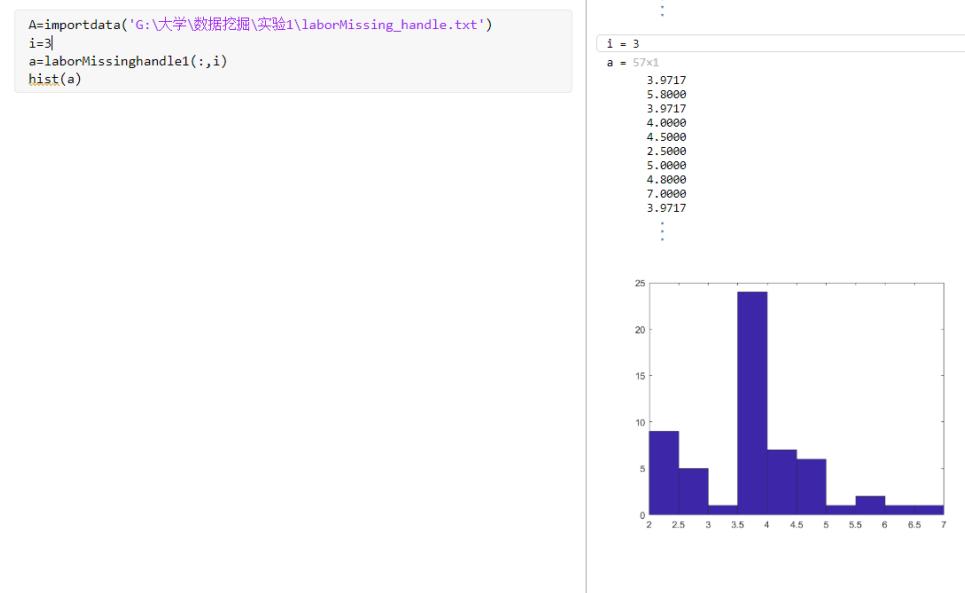


图 9.4.1 可视化所用代码

效果如图：

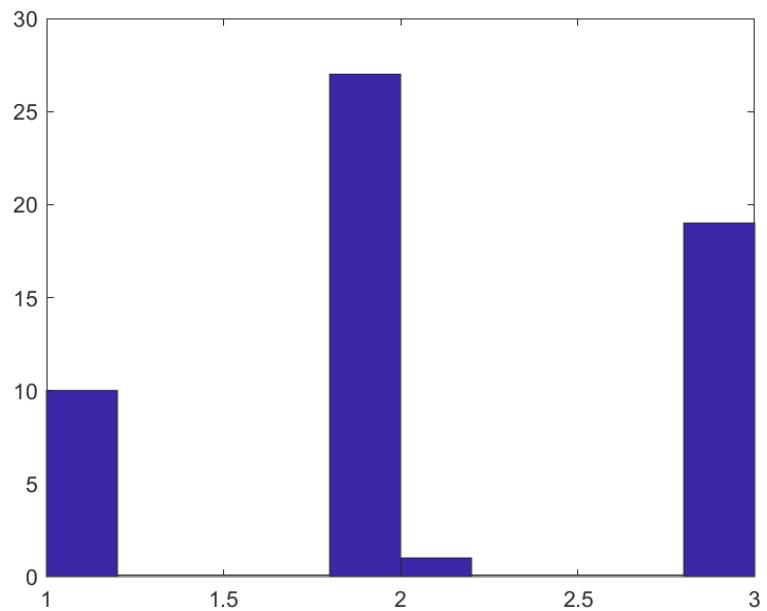


图 9.4.2 第一列 (duration) 数据的直方图

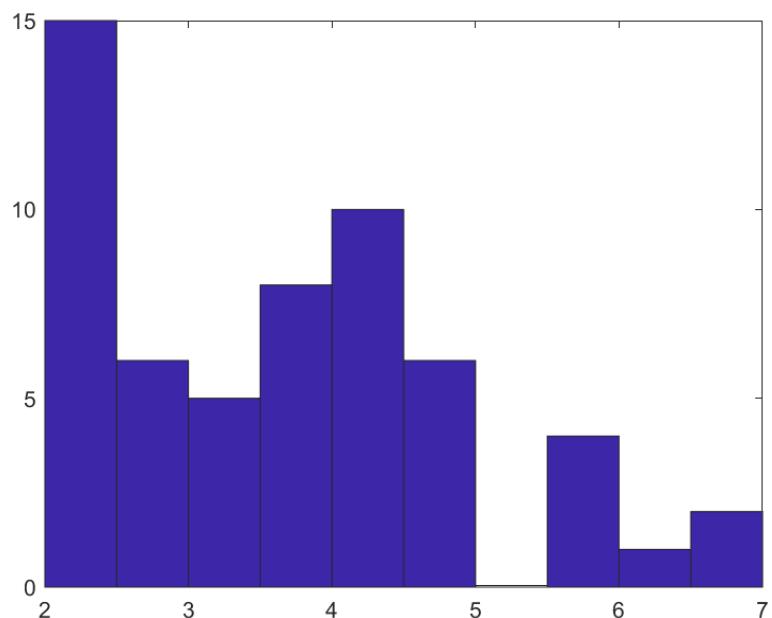


图 9.4.3 第二列 (wage-increase-first-year) 数据的直方图

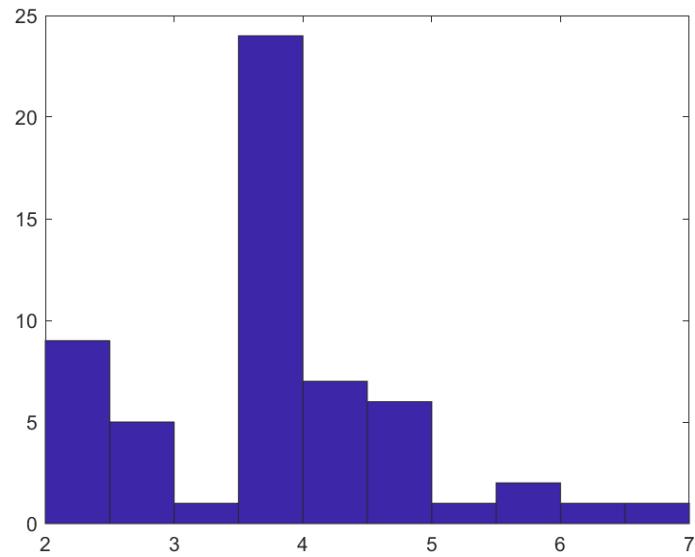


图 9.4.4 第三列（wage-increase-second-year）数据的直方图

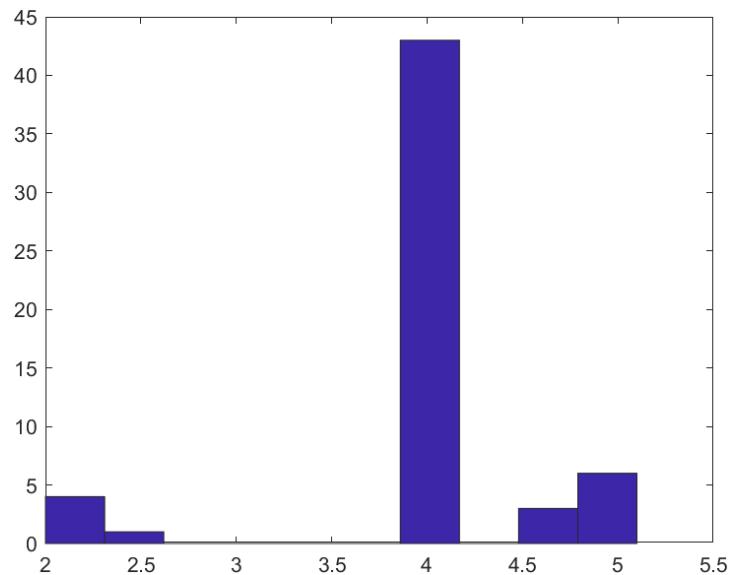


图 9.4.5 第四列（wage-increase-third-year）数据的直方图

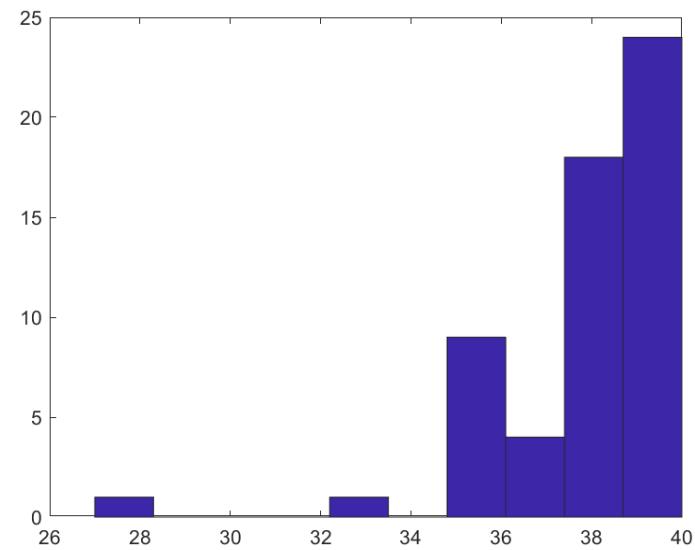


图 9.4.6 第六列 (working-hours) 数据的直方图

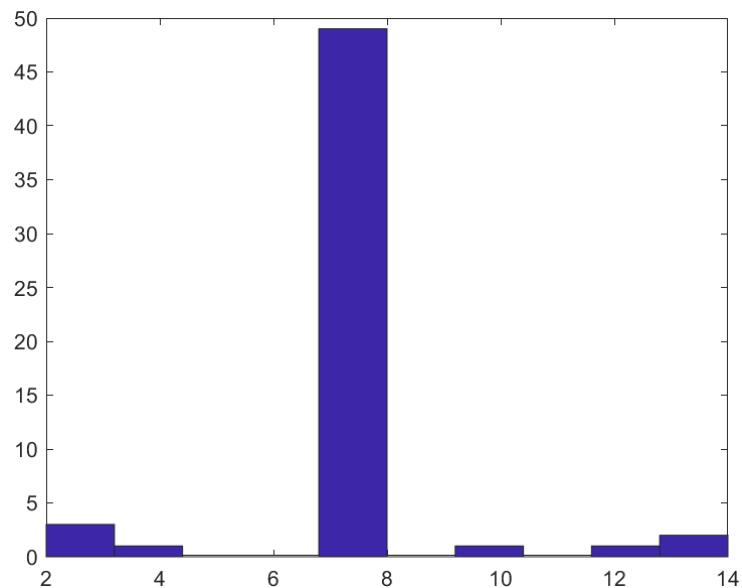


图 9.4.7 第八列 (standby-pay) 数据的直方图

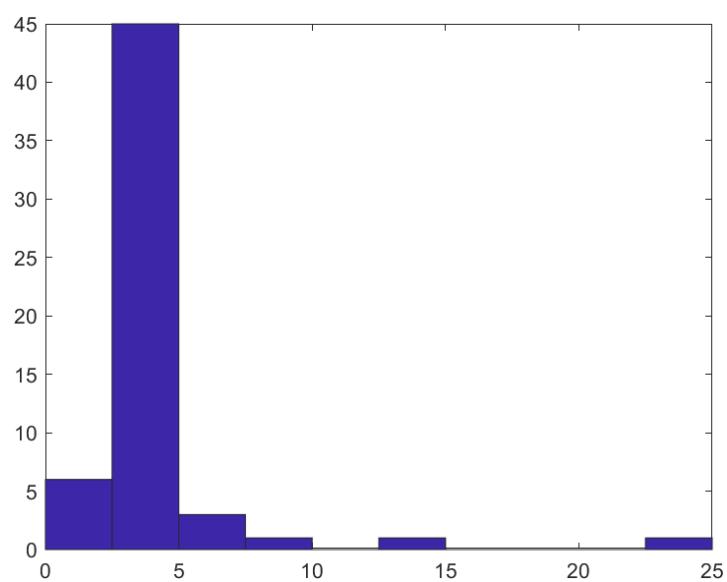


图 9.4.8 第九列 (shift-differential) 数据的直方图

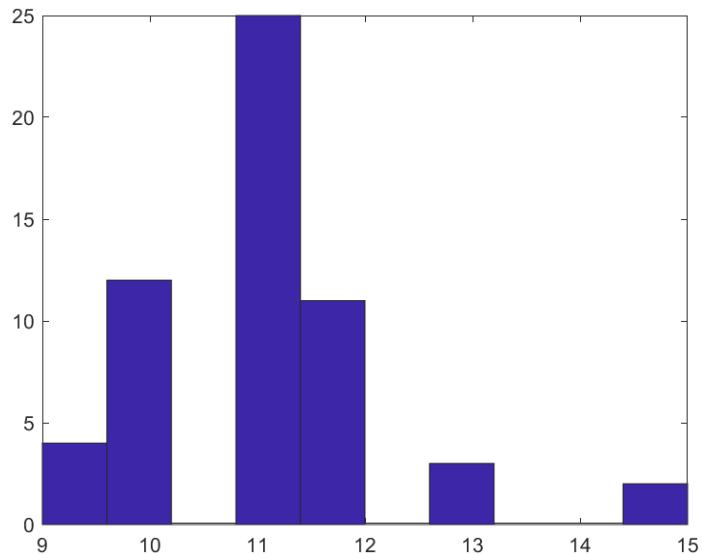


图 9.4.9 第十一列 (statutory-holidays) 数据的直方图

十、实验结论

成功利用 Weka 及 Python 代码对数据集进行归一化、缺失值处理及特征筛选，得到了预期的实验结果，实践了数据清理过程，也完成了数据的可视化。

Weka 的确是一个小巧却功能强大的软件，能对数据进行多种操作，与我用 Python 代码得到的结果大体相似。

十一、总结及心得体会

成功利用 Weka 及 Python 代码对数据集进行归一化、缺失值处理及特征筛选，得到了预期的实验结果，实践了数据清理过程，完成了数据可视化。通过这次实验我对数据的认识更加深刻，并对数据挖掘有了更浓厚的兴趣，希望以后能多进行类似实验。

十二、对本实验过程及方法、手段的改进建议

建议提前发出 PPT 及指导书，以便学生预习。

十三、源码

归一化程序源码：

```

import numpy as np
import pandas as pd

def loadIris(address):#加载数据
    spf=pd.read_csv(address,sep=',',index_col=False,header=None)
    strs=spf[4]
    spf.drop([4],axis=1,inplace=True)
    return spf.values,strs

def normalization(data_matrix):#归一化
    e=1e-5
    for c in range(4):
        maxNum=np.max(data_matrix[:,c])
        minNum=np.min(data_matrix[:,c])
        data_matrix[:,c]=(data_matrix[:,c]-minNum+e)/(maxNum-minNum+e)
    return data_matrix

if __name__=='__main__':#主函数
    filepath='C:\\\\Users\\\\Lenovo\\\\Desktop\\\\dataset\\\\iris.txt'
    writepath='D:\\\\iris_normal.txt'
    data_matrix,str_name=loadIris(filepath)
    data_matrix=normalization(data_matrix)
    spf=pd.DataFrame(data_matrix)
    strs=str_name.values
    spf.insert(4,4,strs)
    spf.to_csv(writepath,index=False,header=False)#将归一化的数据写入 txt

```

缺失值处理程序源码:

```

import pandas as pd
import numpy as np
from collections import Counter


def loadLabor(address): # 加载数据
    spf = pd.read_csv(address, sep=',', index_col=False, header=None)
    column = ['duration', 'wage-increase-first-year', 'wage-increase-second-year',
    'wage-increase-third-year',
    'cost-of-living-adjustment', 'working-hours', 'pension', 'standby-pay',
    'shift-differential', 'education-allowance', 'statutory-holidays',
    'vacation',
    'longterm-disability-assistance', 'contribution-to-dental-plan',
    'bereavement-assistance',
    'contribution-to-health-plan', 'class'] # 所有标签

```

```

spf.columns = column
str_typeName = ['cost-of-living-adjustment', 'pension', 'education-allowance',
                'vacation', 'longterm-disability-assistance',
                'contribution-to-dental-plan',
                'bereavement-assistance', 'contribution-to-health-plan',
                'class'] # 字符数据标签

str2numeric = {}
str2numeric['?'] = '-1'
return spf, str2numeric, str_typeName

def fillMissData(spf, str_typeName): # 缺失值填充
    row, col = spf.shape
    columns = spf.columns
    for column_name in columns: # 将'?'替换为'-1'
        if column_name not in str_typeName:
            tmp = spf[column_name]
            for i in range(len(tmp)):
                if tmp[i] != '?':
                    tmp[i] = float(tmp[i])
            ave = np.average(tmp[tmp != '?'])
            tmp[tmp == '?'] = ave
            spf[column_name] = tmp
        else:
            v = spf[column_name].values
            v1 = v[v != '-1']
            c = Counter(v1)
            cc = c.most_common(1)
            v[v == '-1'] = cc[0][0]
    return spf

if __name__ == '__main__': # 主函数
    filepath = 'G:\\大学\\数据挖掘\\laborMissing.txt'
    fillfilepath = 'G:\\大学\\数据挖掘\\laborMissing_handle.txt'
    spf, str2numeric, str_typeName = loadLabor(filepath)
    spf = fillMissData(spf, str_typeName)
    spf.to_csv(fillfilepath, index=False, header=False)

```

特征筛选程序源码：

```

import pandas as pd
import numpy as np

```

```

from collections import Counter

def loadIris(address):#加载数据
    spf=pd.read_csv(address,sep=',',index_col=False,header=None)
    strs=spf[4]
    spf.drop([4],axis=1,inplace=True) #此处去除了 class 特征
    return spf.values,strs

def getEnergy(c,data,label):#计算增益
    dataLen=len(label)
    energy=0.0
    for key,value in c.items():
        c[key]/=float(dataLen)
        label_picked=label[data==key]
        l=Counter(label_picked)
        e=0.0
        for k,v in l.items():
            r=v/float(value)
            e-=r*np.log2(r)
            energy+=c[key]*e
    return energy

def featureSelection(features,label):#根据增益进行特征筛选
    featureLen=len(features[0,:])
    label_count=Counter(label)
    samples_energy=0.0
    data_len=len(label)
    for i in label_count.keys():
        label_count[i]/=float(data_len)
        samples_energy+=label_count[i]*np.log2(label_count[i])

    informationGain=[]

    for f in range(featureLen):
        af=features[:,f]
        minf=np.min(af)
        maxf=np.max(af)+1e-4
        width=(maxf-minf)/10.0

        d=(af-minf)/width
        dd=np.floor(d)
        c=Counter(dd)

```

```
sub_energy=getEnergy(c,dd,label)

informationGain.append(samples_energy-sub_energy)

return informationGain#返回增益值

if __name__=='__main__':#主函数
    filepath='iris.txt'
    data_matrix,str_name=loadIris(filepath)
    informationGain=featureSelection(data_matrix,str_name.values)
    print(informationGain)
```

报告评分:

指导教师签字:

一、实验室名称：主楼 A2-412

二、实验项目名称：关联规则挖掘

三、实验学时：3

四、实验原理

挖掘关联规则一般步骤

1. 频繁项集产生 (Frequent Itemset Generation)

其目标是发现满足最小支持度阈值的所有项集，这些项集称作频繁项集。

2. 规则的产生 (Rule Generation)

其目标是从上一步发现的频繁项集中提取所有高置信度的规则，这些规则称作强规则 (strong rule)。

写代码部分需要完成以下两个步骤：

1、寻找合适的数据集并进行适当处理

2、根据实验原理，编写 Apriori 算法：

- 1) 读取数据
- 2) 首先获取频繁一项集合（注意对于数据结构的选取提高计算效率）
- 3) 利用 K 项频繁集生成 K+1 频繁候选集：自连接+剪枝
- 4) 重新扫描数据集，确定 K+1 频繁候选集中真正频繁的项集
- 5) 重复上面两步

Apriori 算法

我们有以下性质：

性质一：如果一个项集是频繁的，则它的所有子集一定也是频繁的

性质二：相反，如果一个项集是非频繁的，则它的所有超集也一定是非频繁的

故可以设计以下算法：

- (1) $L_1 = \{\text{频繁 1 项集}\};$
- (2) $\text{for}(k=2; L_{k-1} \neq \emptyset; k++) \text{ do begin}$
- (3) $C_k = \text{apriori_gen}(L_{k-1});$ //新的潜在频繁项集，见下图连接步
- (4) $\text{for all transactions } t \in D \text{ do begin}$
- (5) $C_t = \text{subset}(C_k, t);$ //找出 t 中包含的潜在的频繁项
- (6) $\text{for all candidates } c \in C_t \text{ do}$
- (7) $c.\text{count}++;$
- (8) end
- (9) end
- (10) $L_k = \{c \in C_k | c.\text{count} \geq \text{min_sup}\}$
- (11) $\text{end};$
- (12) $\text{Answer} = \bigcup_k L_k$

```

procedure apriori_gen( $L_{k-1}$ :frequent  $(k-1)$ -itemsets)
(1)   for each itemset  $l_1 \in L_{k-1}$ 
(2)     for each itemset  $l_2 \in L_{k-1}$ 
(3)       if ( $l_1[1] = l_2[1]$ )  $\wedge$  ( $l_1[2] = l_2[2]$ )  $\wedge \dots \wedge$  ( $l_1[k-2] = l_2[k-2]$ )  $\wedge$  ( $l_1[k-1] < l_2[k-1]$ ) then {
(4)          $c = l_1 \bowtie l_2$ ; // join step: generate candidates
(5)         if has_infrequent_subset( $c, L_{k-1}$ ) then
(6)           delete  $c$ ; // prune step: remove unfruitful candidate
(7)         else add  $c$  to  $C_k$ ;
(8)       }
(9)   return  $C_k$ ;

```

1、连接步

图 4.1 连接步代码

其中第(5)步判断子集中是否有不频繁的项集，见下图剪枝步

```

procedure has_infrequent_subset( $c$ : candidate  $k$ -itemset;
 $L_{k-1}$ : frequent  $(k-1)$ -itemsets); // use prior knowledge
(1)   for each  $(k-1)$ -subset  $s$  of  $c$ 
(2)     if  $s \notin L_{k-1}$  then
(3)       return TRUE;
(4)   return FALSE;

```

2、剪枝步

图 4.2 剪枝步代码

下图为 Apriori 算法的一个运行示例。

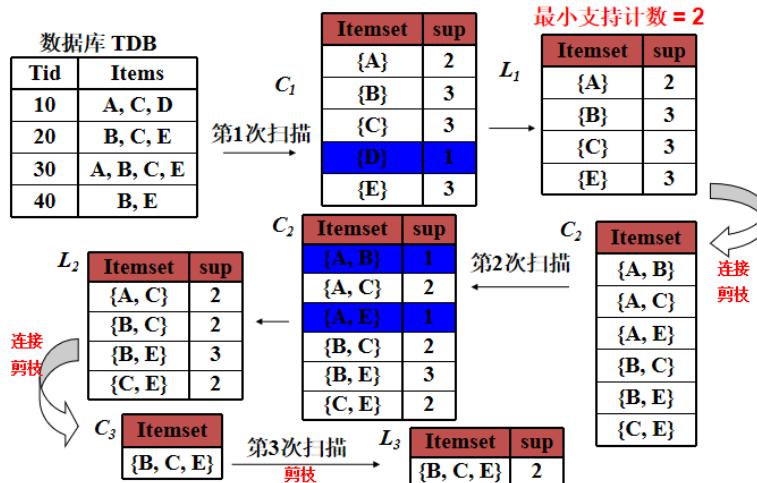


图 4.3 Apriori 算法运行示例

五、实验目的

- 1、掌握关联规则挖掘的基本概念、原理和一般方法
- 2、掌握 Apriori 算法
- 3、了解 FP-GROWTH 算法

六、实验内容：

- (1) 学会调用 WEKA 包实现关联规则的挖掘
- (2) 自己编程实现 Apriori 算法

七、实验器材（设备、元器件）：台式机、笔记本（MagicBook）

八、实验步骤

1. 实现 Apriori 算法

首先认真理解 Apriori 算法。

实现 Apriori 算法的基本流程：

- 1) 首先根据实际问题，获取数据；
- 2) 在此基础上，对数据进行预处理（如有需要）；
- 3) 建立 Apriori 算法类，名称自定，但建议就直接起名为 Apriori；
- 4) 定义 Apriori 的全局变量，在这个算法中，主要涉及到 min_sup 及 min_conf；
- 5) 在主函数里面开始实现算法

频繁项集生成：

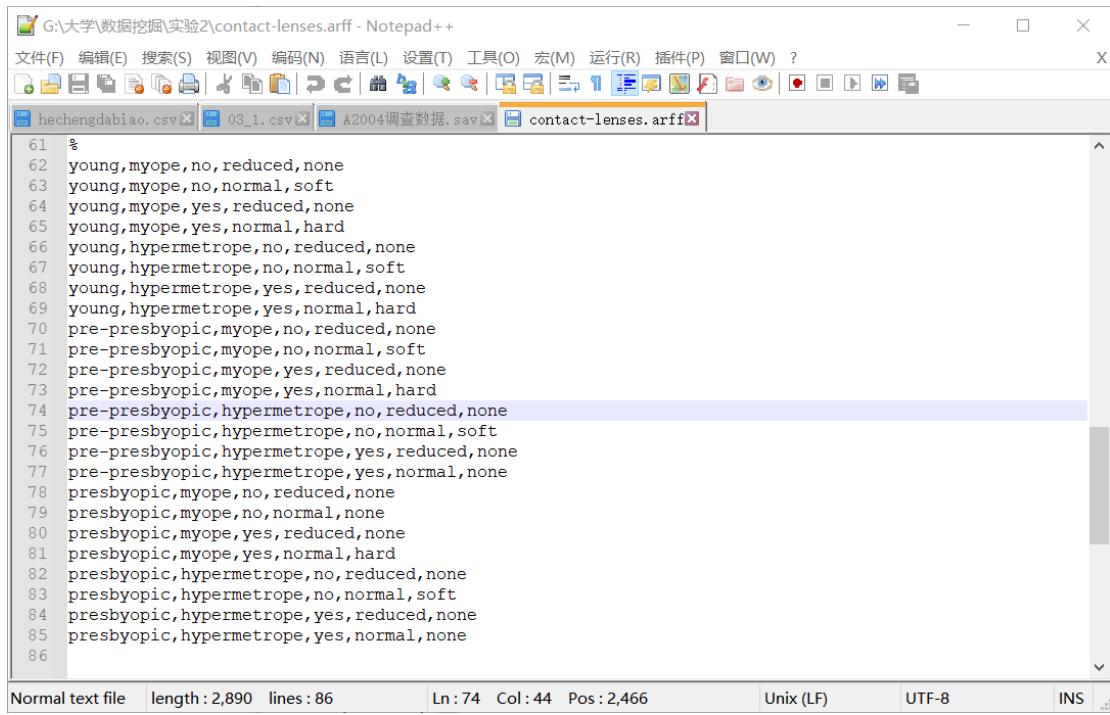
- 1) 读入数据（事务数据库）；
- 2) 找出所有的频繁一项集（遍历整个数据集，找出所有一项集，并计算计数(函数 scanFirDatas())，如果 support>min_sup，则将其放入频繁一项集的变量中(函数 getFreSet)。在此基础上，构建 K 项集 ($K > 1$)
- 3) 在 $K-1$ 项集的基础上生成 K 项候选集，包括自连接步和剪枝步。其中自连接步将任意两个项集如果 $K-2$ 项相同，剩余的一项不同的两个项集连接（函数 selfConn）。其中，剪枝步（函数 cutBranch）就是将自连接生成的候选集的任意子集在 $K-1$ 项中是否存在，否则删除。
- 4) 产生获选 K 项集后，遍历整个数据集，检查这些候选集是否频繁，如果是，加入 K 项频繁项集中。循环这个过程，直到找到的 K 频繁项集为空，退出（函数 algorithm 中的 while 循环）。

关联规则获取（函数 genAssRule）： 在找到频繁项集的基础上，将 $K > 1$ 的任意频繁项集分成任意的两子集 A 和 B，然后计算 $A \rightarrow B$ 及 $B \rightarrow A$ 的置信度（函数 calcConf）。如果置信度 $> \text{min_conf}$ ，那么这些规则是强规则，并保存起来。

九、实验数据及结果分析

9.1 使用 WEKA 图形界面实现关联规则挖掘

打开 Weka，打开文件 contact-lenses.arff。contact-lenses.arff 数据如图所示：



The screenshot shows a Notepad++ window with the title "G:\大学\数据挖掘\实验2\contact-lenses.arff - Notepad++". The file contains 86 lines of data representing contact lens usage based on various factors. The data starts with a percentage sign (%) at line 61 and continues with specific attribute-value pairs for each observation from line 62 to 86.

```
61 %
62 young,myope,no,reduced,none
63 young,myope,no,normal,soft
64 young,myope,yes,reduced,none
65 young,myope,yes,normal,hard
66 young,hypermetrope,no,reduced,none
67 young,hypermetrope,no,normal,soft
68 young,hypermetrope,yes,reduced,none
69 young,hypermetrope,yes,normal,hard
70 pre-presbyopic,myope,no,reduced,none
71 pre-presbyopic,myope,no,normal,soft
72 pre-presbyopic,myope,yes,reduced,none
73 pre-presbyopic,myope,yes,normal,hard
74 pre-presbyopic,hypermetrope,no,reduced,none
75 pre-presbyopic,hypermetrope,no,normal,soft
76 pre-presbyopic,hypermetrope,yes,reduced,none
77 pre-presbyopic,hypermetrope,yes,normal,none
78 presbyopic,myope,no,reduced,none
79 presbyopic,myope,no,normal,none
80 presbyopic,myope,yes,reduced,none
81 presbyopic,myope,yes,normal,hard
82 presbyopic,hypermetrope,no,reduced,none
83 presbyopic,hypermetrope,no,normal,soft
84 presbyopic,hypermetrope,yes,reduced,none
85 presbyopic,hypermetrope,yes,normal,none
86
```

Normal text file | length : 2,890 | lines : 86 | Ln : 74 | Col : 44 | Pos : 2,466 | Unix (LF) | UTF-8 | INS

图 9.1.1 数据集一览

Weka 选项卡中选择 associate，associator 选择 Apriorio，参数设置如下：

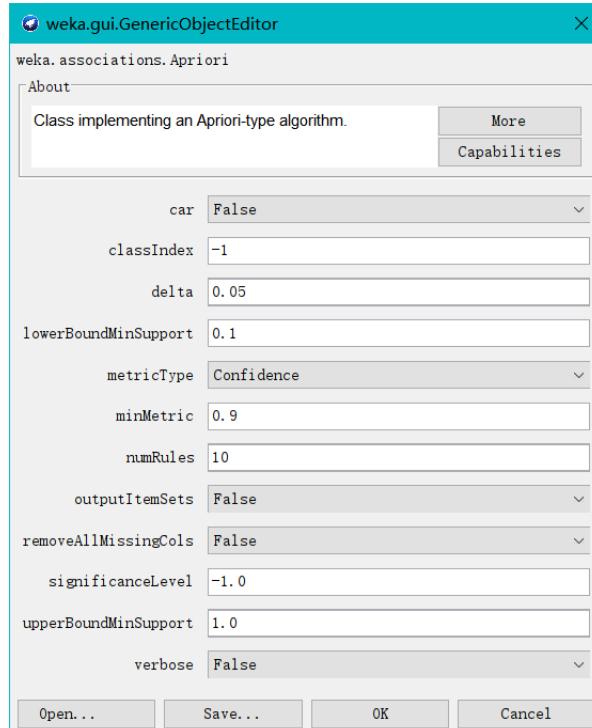


图 9.1.2 Weka 关联规则挖掘时的参数设置

(可以看到最小支持度 0.1，最小置信度 0.9)

然后点击 start 开始挖掘关联规则，运行结果如图：

```
Associate output
Result list (right)
20/29/0 -> Apriori
Scheme: weka.associations.Apriori -N 10 -T 0 -C 0.9 -D 0.05 -U 1.0 -M 0.1 -S -1.0 -c -1
Relation: contact-lenses
Instances: 24
Attributes: age
spectacle-prescrip
astigmatism
tear-prod-rate
contact-lenses
*** Associate model (full training set) ***
Apriori
=====
Minimum support: 0.2 (5 instances)
Minimum metric <confidence>: 0.9
Number of cycles performed: 16
Generated sets of large itemsets:
Size of set of large itemsets L(1): 11
Size of set of large itemsets L(2): 21
Size of set of large itemsets L(3): 6
Best rules found:
1. tear-prod-rate=reduced 12 ==> contact-lenses=none 12 conf:(1)
2. spectacle-prescrip=myope tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1)
3. spectacle-prescrip=hypermetropic tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1)
4. astigmatism=no tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1)
5. astigmatism=yes tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1)
6. contact-lenses=soft 5 ==> astigmatism=no 5 conf:(1)
7. contact-lenses=soft 5 ==> tear-prod-rate=normal 5 conf:(1)
8. tear-prod-rate=normal contact-lenses=soft 5 ==> astigmatism=no 5 conf:(1)
9. astigmatism=no contact-lenses=soft 5 ==> tear-prod-rate=normal 5 conf:(1)
10. contact-lenses=soft 5 ==> astigmatism=no tear-prod-rate=normal 5 conf:(1)
```

图 9.1.3 Weka 关联规则挖掘时的运行结果

可见共有 10 条关联规则被挖掘出来：

1. tear-prod-rate=reduced 12 ==> contact-lenses=none 12 conf:(1)
2. spectacle-prescrip=myope tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1)
3. spectacle-prescrip=hypermetropic tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1)
4. astigmatism=no tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1)
5. astigmatism=yes tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1)
6. contact-lenses=soft 5 ==> astigmatism=no 5 conf:(1)
7. contact-lenses=soft 5 ==> tear-prod-rate=normal 5 conf:(1)
8. tear-prod-rate=normal contact-lenses=soft 5 ==> astigmatism=no 5 conf:(1)
9. astigmatism=no contact-lenses=soft 5 ==> tear-prod-rate=normal 5 conf:(1)
10. contact-lenses=soft 5 ==> astigmatism=no tear-prod-rate=normal 5 conf:(1)

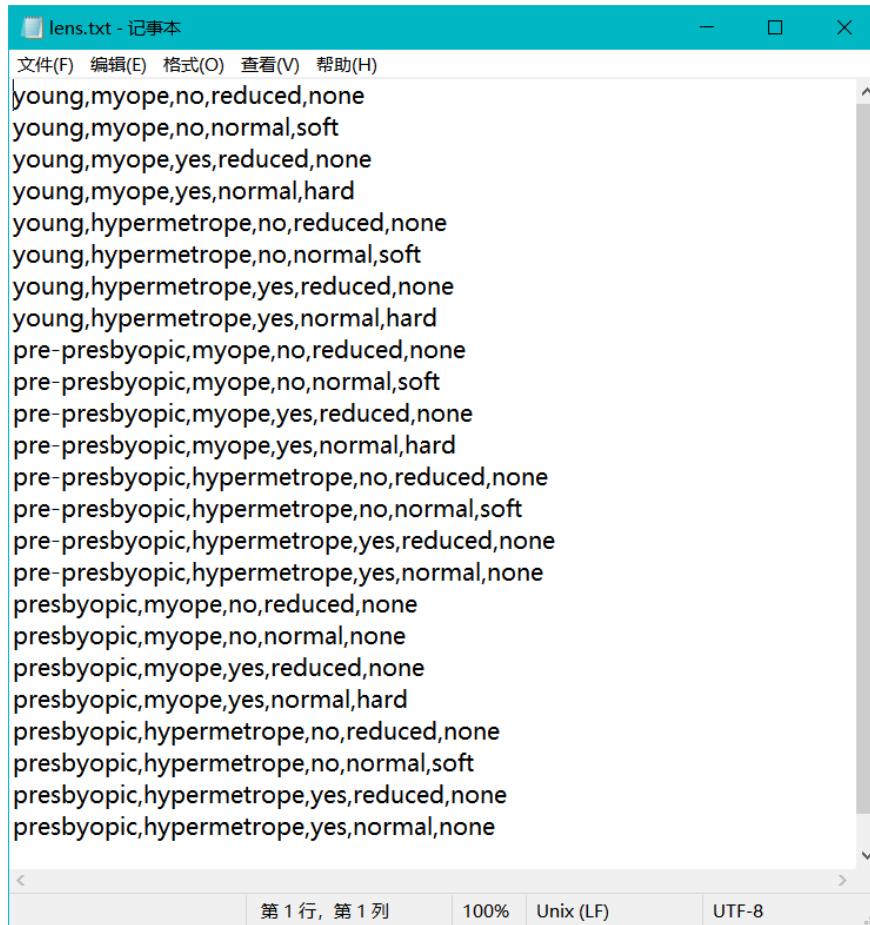
9.2 自己动手实现 Apriori 算法

9.2.1 算法设计

- 1) 首先根据实际问题，获取数据：

简易数据集如 python 文件所示： traDatas=['abc','ae','abc','ade'];

复杂数据集为 contact-lenses.arff，提取其中数据至 lens.txt，如下图所示：



```
lens.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
young,myope,no,reduced,none
young,myope,no,normal,soft
young,myope,yes,reduced,none
young,myope,yes,normal,hard
young,hypermetrope,no,reduced,none
young,hypermetrope,no,normal,soft
young,hypermetrope,yes,reduced,none
young,hypermetrope,yes,normal,hard
pre-presbyopic,myope,no,reduced,none
pre-presbyopic,myope,no,normal,soft
pre-presbyopic,myope,yes,reduced,none
pre-presbyopic,myope,yes,normal,hard
pre-presbyopic,hypermetrope,no,reduced,none
pre-presbyopic,hypermetrope,no,normal,soft
pre-presbyopic,hypermetrope,yes,reduced,none
pre-presbyopic,hypermetrope,yes,normal,none
presbyopic,myope,no,reduced,none
presbyopic,myope,no,normal,none
presbyopic,myope,yes,reduced,none
presbyopic,myope,yes,normal,hard
presbyopic,hypermetrope,no,reduced,none
presbyopic,hypermetrope,no,normal,soft
presbyopic,hypermetrope,yes,reduced,none
presbyopic,hypermetrope,yes,normal,none
```

图 9.2.1.1 复杂数据集

2) 在此基础上, 对数据进行预处理(如有需要);

针对复杂数据集有必要对数据预处理, 建立各单词与字母之间的对应关系即可, 如“young”对应“a”, “myope”对应“b”等等。

3) 建立 Apriori 算法类, 名称自定, 但建议就直接起名为 Apriori;

```
class Apriori:
    traDatas=[]
    traLen=0
    k=1
    traCount={}
    freTran={}          #k一定时的频繁项集
    sup=0
    conf=0
    freAllTran={}      #所有频繁项集
    rules=[]
```

图 9.2.1.2 Apriori 类

其中包括多个量和函数, 如 traDatas(变量)、scanFirDatas(函数)等。

其中比较重要的函数有:

```

def getFreset(self):          #得到频繁项集及其频数，此键值对存在freAllTran中
    self.freTran={}
    for tra in self.traCount.keys():
        if self.traCount[tra]>=self.sup and len(tra)==self.k:
            self.freTran[tra]=self.traCount[tra]
            self.freAllTran[tra]=self.traCount[tra]

```

图 9.2.1.3 getFreset 函数

此函数得到频繁项集及其频数，此键值对存在 freAllTran（所有频繁的项集）中。

```

def selfConn(self):           #自连接，从k元频繁项集生成k+1元频繁项集
    self.traCount={}
    for item in itertools.combinations(self.freTran.keys(),2):   #生成所有k元频繁项集的两两组合
        if self.getCmpTwoSet(item[0],item[1]) == True:
            key=''.join(sorted(''.join(set(item[0]).union(set(item[1]))))) #key为满足条件的k+1元频繁项集（候选）
            if self.cutBranch(key) != False:                                #如果key是频繁项集，则先将其对应的频数置为0（创建键值对）
                self.traCount[key]=0

```

图 9.2.1.4 selfConn 函数

此函数实现自连接，从 k 元频繁项集生成 k+1 元频繁项集。

```

def cutBranch(self,key):      #剪枝，如果k+1元候选频繁项集有非频繁的k项子集，则返回False
    for subKey in list(itertools.combinations(key,self.k)):
        if ''.join(list(subKey)) not in self.freTran.keys():
            return False

```

图 9.2.1.5 cutBranch 函数

此函数实现剪枝，如果 k+1 元候选频繁项集有非频繁的 k 项子集，则返回 False，提高 k+1 元频繁项集生成效率。

```

def genAssRule(self):          #产生关联规则
    container=[]
    ruleSet=set()              #关联规则的集合
    for item in self.freTran.keys():
        self.permutation(item,'',container)
    for item in container:
        for i in range(1,len(item)):
            #print(item[:i]+''+item[i:])
            ruleSet.add('''.join(sorted(item[:i]))''.join(sorted(item[i:]))) #在关联规则集合中添加规则（键值对）
    for rule in ruleSet:
        if self.calcConf(rule[0],rule[1])>self.conf:                      #打印置信度大于最小置信度的规则
            #continue
            if rule[0] + "---->>" + rule[1] not in self.rules:
                self.rules.append(rule[0] + "---->>" + rule[1])
            #print(rule[0] + "---->>" + rule[1])

```

图 9.2.1.6 genAssRule 函数

此函数用于产生关联规则，将满足条件的规则存在 rules 中。

```

def algorithm(self): #Apriori算法
    self.scanFirDatas()
    while self.traCount != {}:
        self.getReset()
        self.selfConn()
        self.scanDatas()
        #print(self.freAllTran)
        #print(self.freTran)
        self.genAssRule()
        print(self.rules)

```

图 9.2.1.6 Apriori 函数

此函数为核心函数，完成整个 Apriori 算法。

4) 定义 Apriori 的全局变量，在这个算法中，主要涉及到 min_sup 及 min_conf;

```

def __init__(self, traDatas, sup, conf): #初始化，将数据、最小支持度、最小置信度赋给生成的Apriori类
    self.traDatas=traDatas
    self.traLen=len(traDatas)
    self.sup=sup
    self.conf=conf

```

图 9.2.1.7 参数

5) 在主函数里面开始实现算法

```

def algorithm(self): #Apriori算法
    self.scanFirDatas()
    while self.traCount != {}:
        self.getReset()
        self.selfConn()
        self.scanDatas()
        #print(self.freAllTran)
        #print(self.freTran)
        self.genAssRule()
        print(self.rules)

```

图 9.2.1.7 主函数

此函数为主函数，完成整个 Apriori 算法。

9.2.2 针对简易数据集的测试

简易数据集如 python 文件所示： traDatas=['abc','ae','abc','ade']

```
traDatas=['abc','ae','abc','ade'] #使用的数据集，用于挖掘关联规则
```

图 9.2.2.1 简易数据集

运行程序得到如下结果：

The screenshot shows the PyCharm IDE interface. The top bar has 'File', 'Edit', 'View', 'Navigate', 'Code', 'Refactor', 'Run', 'Tools', 'VCS', 'Window', 'Help' menus. The title bar says '实验2 association.py'. The left sidebar shows a project tree with 'association.py' selected. The main code editor contains Python code for an Apriori algorithm. Line 105 defines the data set: `traDatas=['abc','ae','abc','ade'] #使用的数据集，用于挖掘关联规则`. Line 106 calls `aprioriApriori(traDatas, 2, 0.7)`. Line 107 calls `apriori.algorithm()`. The bottom 'Run' tab shows the command `C:\Users\Jerry\AppData\Local\Programs\Python\Python38\python.exe 6:/大学/数据挖掘/实验2\association.py` and the output: `['ab---->>>c', 'ac---->>b', 'b---->>>ac', 'bc---->>>a', 'c---->>>ab']`. The status bar at the bottom right indicates '104:26 CRLF UTF-8 4 spaces Python 3.8 (3)'.

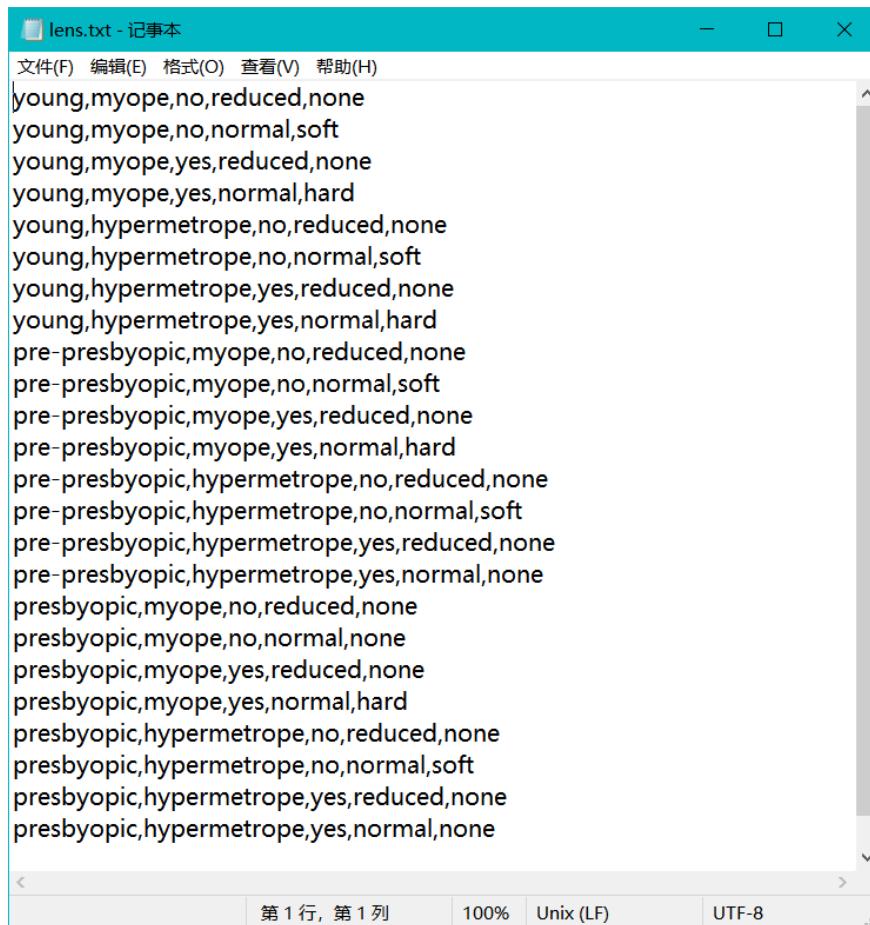
图 9.2.2.2 简易数据集的运行结果

可见输出了 5 条关联规则：

```
['ab---->>>c', 'ac---->>b', 'b---->>>ac', 'bc---->>>a', 'c---->>>ab']
```

9.2.3 针对复杂数据集的测试

复杂数据集为 contact-lenses.arff，提取其中数据至 lens.txt，如下图所示：



```
lens.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
young,myope,no,reduced,none
young,myope,no,normal,soft
young,myope,yes,reduced,none
young,myope,yes,normal,hard
young,hypermetrope,no,reduced,none
young,hypermetrope,no,normal,soft
young,hypermetrope,yes,reduced,none
young,hypermetrope,yes,normal,hard
pre-presbyopic,myope,no,reduced,none
pre-presbyopic,myope,no,normal,soft
pre-presbyopic,myope,yes,reduced,none
pre-presbyopic,myope,yes,normal,hard
pre-presbyopic,hypermetrope,no,reduced,none
pre-presbyopic,hypermetrope,no,normal,soft
pre-presbyopic,hypermetrope,yes,reduced,none
pre-presbyopic,hypermetrope,yes,normal,none
presbyopic,myope,no,reduced,none
presbyopic,myope,no,normal,none
presbyopic,myope,yes,reduced,none
presbyopic,myope,yes,normal,hard
presbyopic,hypermetrope,no,reduced,none
presbyopic,hypermetrope,no,normal,soft
presbyopic,hypermetrope,yes,reduced,none
presbyopic,hypermetrope,yes,normal,none
```

图 9.2.3.1 复杂数据集

针对复杂数据集我们有必要对数据预处理，建立各单词与字母之间的对应关系即可，如“young”对应“a”，“myope”对应“b”等等。实际操作过程中设计算法如下：

```

f = open("G:\\大学\\数据挖掘\\实验2\\lens.txt", encoding='utf-8')
data = []
while True:                                #将lens.txt文件读入，每行作为一个字符串
    line = f.readline()
    if line:
        index = line.find('\n')
        listline = list(line)
        del(listline[index:index+2])
        data.append(''.join(listline))
    else:
        break
f.close()
for i in range(len(data)):                  #将每行的字符串用“,”隔开，识别不同属性
    data[i] = data[i].split(sep=',')
variable = []
for i in data:                                #将所有出现的属性存在列表中
    for j in i:
        if j not in variable:
            variable.append(j)

letter_word = dict()
word_letter = dict()
for i in range(len(variable)):               #建立字母与单词的对应关系，于是我们就能调用Apriori算法了
    word_letter[variable[i]] = chr(ord('a') + i)
    letter_word[chr(ord('a') + i)] = variable[i]
traDatas = []
for i in data:
    string = ''
    for j in i:
        string = string + ''.join(word_letter[j])
    traDatas.append(''.join(sorted(string)))

```

图 9.2.3.2 复杂数据集的预处理算法

其次对 Apriori 类作了些许修改：

```

import itertools

class Apriori:
    traDatas = []
    traLen = 0
    k = 1
    traCount = {}
    freTran = {}           #k一定时的频数项集
    sup = 0
    conf = 0
    freAllTran = {}       #所有频数项集
    rules = []
    letter_word = dict()

    def __init__(self, traDatas, sup, conf, letter_word): #初始化，将数据、最小支持度、最小置信度赋给生成的Apriori类
        self.traDatas = traDatas
        self.traLen = len(traDatas)
        self.sup = sup
        self.conf = conf

```

加入了 letter_word 参数，在外部调用时输入，为字母与单词的对应关系。

接下来运行代码，可以得到以下结果：

```
C:\Users\Jerry\AppData\Local\Programs\Python\Python38\python.exe G:/大学/数据挖掘/实验2/assocation2.py
myope,no,reduced---->>>none
myope,reduced,yes---->>>none
myope,none,yes---->>>reduced
myope,normal,yes---->>>hard
myope,normal,hard---->>>yes
myope,yes,hard---->>>normal
myope,hard---->>>normal,yes
no,reduced,hypermetrope---->>>none
no,none,hypermetrope---->>>reduced
no,normal,hypermetrope---->>>soft
no,soft,hypermetrope---->>>normal
normal,soft,hypermetrope---->>>no
soft,hypermetrope---->>>no,normal
reduced,yes,hypermetrope---->>>none

Process finished with exit code 0
```

图 9.2.3.3 运行结果：产生的关联规则

所有的关联规则如下：

```
myope,no,reduced---->>>none
myope,reduced,yes---->>>none
myope,none,yes---->>>reduced
myope,normal,yes---->>>hard
myope,normal,hard---->>>yes
myope,yes,hard---->>>normal
myope,hard---->>>normal,yes
no,reduced,hypermetrope---->>>none
no,none,hypermetrope---->>>reduced
no,normal,hypermetrope---->>>soft
no,soft,hypermetrope---->>>normal
normal,soft,hypermetrope---->>>no
soft,hypermetrope---->>>no,normal
reduced,yes,hypermetrope---->>>none
```

共 14 条关联规则。对比 9.1，可以看出 Python 运行结果与 9.1 基于 Weka 的关联规则挖掘得到结果几乎完全一致。

```
Best rules found:

1. tear-prod-rate=reduced 12 ==> contact-lenses=none 12      conf: (1)
2. spectacle-prescrip=myope tear-prod-rate=reduced 6 ==> contact-lenses=none 6      conf: (1)
3. spectacle-prescrip=hypermetrope tear-prod-rate=reduced 6 ==> contact-lenses=none 6      conf: (1)
4. astigmatism=no tear-prod-rate=reduced 6 ==> contact-lenses=none 6      conf: (1)
5. astigmatism=yes tear-prod-rate=reduced 6 ==> contact-lenses=none 6      conf: (1)
6. contact-lenses=soft 5 ==> astigmatism=no 5      conf: (1)
7. contact-lenses=soft 5 ==> tear-prod-rate=normal 5      conf: (1)
8. tear-prod-rate=normal contact-lenses=soft 5 ==> astigmatism=no 5      conf: (1)
9. astigmatism=no contact-lenses=soft 5 ==> tear-prod-rate=normal 5      conf: (1)
10. contact-lenses=soft 5 ==> astigmatism=no tear-prod-rate=normal 5      conf: (1)
```

```
C:\Users\Jerry\AppData\Local\Programs\Python\Python38\python.exe G:/大学/数据挖掘/实验2/assoiacion2.py
myope,no,reduced---->>>none
myope,reduced,yes---->>>none
myope,none,yes---->>>reduced
myope,normal,yes---->>>hard
myope,normal,hard---->>>yes
myope,yes,hard---->>>normal
myope,hard---->>>normal,yes
no,reduced,hypermetrope---->>>none
no,none,hypermetrope---->>>reduced
no,normal,hypermetrope---->>>soft
no,soft,hypermetrope---->>>normal
normal,soft,hypermetrope---->>>no
soft,hypermetrope---->>>no,normal
reduced,yes,hypermetrope---->>>none

Process finished with exit code 0
```

图 9.2.3.4 Weka、Python 运行结果对比
若将参数调成一样，理论上能得到一致的结果。

十、实验结论

成功利用 Weka 及 Python 代码对数据集进行归一化、缺失值处理及特征筛选，得到了预期的实验结果，实践了数据清理过程。

Weka 的确是一个小巧却功能强大的软件，能对数据进行多种操作，与我用 Python 代码得到的结果大体相似。

Apriori 算法的确是挖掘关联规则的一种可行、有效的算法

十一、总结及心得体会

成功利用 Weka 及 Python 代码对数据集进行关联规则的测试，并用 Python 代码实现了简单数据集和复杂数据集的关联规则挖掘，得到了预期的实验结果。通过这次实验我对数据的认识更加深刻，提高了动手能力，也更加理解了 Apriori 算法、FP 增长算法，并对数据挖掘有了更浓厚的兴趣，希望以后能多进行类似实验。

十二、对本实验过程及方法、手段的改进建议

建议提前发出 PPT 及指导书，以便学生预习。

十三、源码

针对简单数据集：

```

import collections
import itertools

class Apriori:
    traDatas=[]
    traLen=0
    k=1
    traCount={}
    freTran={} #k 一定时的频繁项集
    sup=0
    conf=0
    freAllTran={} #所有频繁项集
    rules=[]

    def __init__(self,traDatas,sup,conf): #初始化，将数据、最小支持度、最小置信度赋给生成的Apriori类
        self.traDatas=traDatas
        self.traLen=len(traDatas)
        self.sup=sup
        self.conf=conf

    def scanFirDatas(self): #扫描数据，将数据中所有字母及其频数的键值对存在字典类型的traCount中
        tmpStr="".join(traDatas)
        self.traCount=dict(collections.Counter(tmpStr))
        return self.traCount

    def getFreset(self): #得到频繁项集及其频数，此键值对存在freAllTran中
        self.freTran={}
        for tra in self.traCount.keys():
            if self.traCount[tra]>=self.sup and len(tra)==self.k:
                self.freTran[tra]=self.traCount[tra]
                self.freAllTran[tra]=self.traCount[tra]

    def cmpTwoSet(self,setA,setB): #判断两个k元集的并是否恰为k+1元集(众所周知，如果|A-B|=|B-A|=1，则k元集A、B的并恰为k+1元集)
        setA=set(setA)
        setB=set(setB)
        if len(setA-setB)==1 and len(setB-setA)==1:
            return True
        else:
            return False

```

```

def selfConn(self):                                #自连接，从 k 元频繁项集生成 k+1 元频繁项
集
    self.traCount={}
    for item in itertools.combinations(self.freTran.keys(),2):      #生成所有 k 元频繁
项集的两两组合
        if self.getCmpTwoSet(item[0],item[1]) == True:
            key="join(sorted("join(set(item[0]).union(set(item[1])))))"  #key
为满足条件的 k+1 元频繁项集（候选）
            if self.cutBranch(key) != False:                               #如果 key 是频
繁项集，则先将其对应的频数置为 0（创建键值对）
                self.traCount[key]=0

def scanDatas(self):                                #扫描所有数据（事务），更新新产生 key 对应
的频数
    self.k=self.k+1
    for tra in traDatas:
        for key in self.traCount.keys():
            self.traCount[key]=self.traCount[key]+self.findChars(tra,key)

def cutBranch(self,key):                           #剪枝，如果 k+1 元候选频繁项集有非频繁的 k
项子集，则返回 False
    for subKey in list(itertools.combinations(key,self.k)):
        if "join(list(subKey))" not in self.freTran.keys():
            return False

def findChars(self,str(chars):                  #如果 chars 所有字母都在 str 中，则返回 1，否
则返回 0
    for char in list(chars):
        if char not in str:
            return 0
    return 1

def permutation(self,string,pre_str,container):
    if len(string)==1:
        container.append(pre_str+string)

    for idx,str in enumerate(string):          #生成 string 的索引序列
        new_str=string[:idx]+string[idx+1:]
        new_pre_str=pre_str+str

        self.permutation(new_str,new_pre_str,container)

```

```

def genAssRule(self):          #产生关联规则
    container=[]
    ruleSet=set()           #关联规则的集合
    for item in self.freTran.keys():
        self.permutation(item,container)
    for item in container:
        for i in range(1,len(item)):
            #print(item[:i]+" "+item[i:])
            ruleSet.add(("".join(sorted(item[:i])),"".join(sorted(item[i:])))) #在关联规则集合中添加规则（键值对）
    for rule in ruleSet:
        if self.calcConfi(rule[0],rule[1])>self.conf:
#打印置信度大于最小置信度的规则
            #continue
            if rule[0]+">>>>"+rule[1] not in self.rules:
                self.rules.append(rule[0]+">>>>"+rule[1])
            #print(rule[0]+">>>>"+rule[1])

def calcConfi(self,first,last):      #计算置信度
    return
self.freAllTran["".join(sorted((first+last)))]/self.freAllTran["".join(sorted(first))]

def algorithm(self):               #Apriori 算法
    self.scanFirDatas()
    while self.traCount != {}:
        self.getFreset()
        self.selfConn()
        self.scanDatas()
        #print(self.freAllTran)
        #print(self.freTran)
        self.genAssRule()
        print(self.rules)

traDatas=['abc','ae','abc','ade']      #使用的数据集，用于挖掘关联规则
apriori=Apriori(traDatas,2,0.7)
apriori.algorithm()

```

针对复杂数据集

```

import collections
import itertools

class Apriori:
    traDatas=[]

```

```

traLen=0
k=1
traCount={}
freTran={} #k一定时的频繁项集
sup=0
conf=0
freAllTran={} #所有频繁项集
rules=[]
letter_word=dict()

def __init__(self,traDatas,sup,conf,letter_word): #初始化，将数据、最小支持度、最小置信度赋给生成的 Apriori 类
    self.traDatas=traDatas
    self.traLen=len(traDatas)
    self.sup=sup
    self.conf=conf

def scanFirDatas(self): #扫描数据，将数据中所有字母及其频数的键值对存在字典类型的 traCount 中
    tmpStr=".join(traDatas)"
    self.traCount=dict(collections.Counter(tmpStr))
    return self.traCount

def getFreset(self): #得到频繁项集及其频数，此键值对存在 freAllTran 中
    self.freTran={}
    for tra in self.traCount.keys():
        if self.traCount[tra]>=self.sup and len(tra)==self.k:
            self.freTran[tra]=self.traCount[tra]
            self.freAllTran[tra]=self.traCount[tra]

def cmpTwoSet(self,setA,setB): #判断两个 k 元集的并是否恰为 k+1 元集（众所周知，如果|A-B|=|B-A|=1，则 k 元集 A、B 的并恰为 k+1 元集）
    setA=set(setA)
    setB=set(setB)
    if len(setA-setB)==1 and len(setB-setA)==1:
        return True
    else:
        return False

def selfConn(self): #自连接，从 k 元频繁项集生成 k+1 元频繁项集
    self.traCount={}

```

```

        for item in itertools.combinations(self.freTran.keys(),2):    #生成所有 k 元频繁
项集的两两组合
            if self.getCmpTwoSet(item[0],item[1]) == True:
                key="".join(sorted("".join(set(item[0]).union(set(item[1])))))  #key
为满足条件的 k+1 元频繁项集（候选）
            if self.cutBranch(key) != False:                                #如果 key 是频
繁项集，则先将其对应的频数置为 0（创建键值对）
                self.traCount[key]=0

def scanDatas(self):                                         #扫描所有数据（事务），更新新产生 key 对应
的频数
    self.k=self.k+1
    for tra in traDatas:
        for key in self.traCount.keys():
            self.traCount[key]=self.traCount[key]+self.findChars(tra,key)

def cutBranch(self,key):                                     #剪枝，如果 k+1 元候选频繁项集有非频繁的 k
项子集，则返回 False
    for subKey in list(itertools.combinations(key,self.k)):
        if "".join(list(subKey)) not in self.freTran.keys():
            return False

def findChars(self,str(chars):                                #如果 chars 所有字母都在 str 中，则返回 1，否
则返回 0
    for char in list(chars):
        if char not in str:
            return 0
    return 1

def permutation(self,string,pre_str,container):
    if len(string)==1:
        container.append(pre_str+string)

    for idx,str in enumerate(string):      #生成 string 的索引序列
        new_str=string[:idx]+string[idx+1:]
        new_pre_str=pre_str+str

        self.permutation(new_str,new_pre_str,container)

def genAssRule(self):                                       #产生关联规则
    container=[]
    ruleSet=set()                                         #关联规则的集合
    for item in self.freTran.keys():

```

```

        self.permutation(item,"container")
    for item in container:
        for i in range(1,len(item)):
            #print(item[:i]+" "+item[i:])
            ruleSet.add(("join(sorted(item[:i]))","join(sorted(item[i:]))")) #在关联规则集合中添加规则（键值对）
    for rule in ruleSet:
        if self.calcConf(rule[0],rule[1])>self.conf:
#打印置信度大于最小置信度的规则
            #continue
            temp='join([letter_word[i] for i in rule[0]]) + '---->>>
+'join([letter_word[i] for i in rule[1]])'
            if temp not in self.rules:
                self.rules.append(temp)
            #print(rule[0]+---->>>+rule[1])

def calcConf(self,first,last):                      #计算置信度
    return
self.freAllTran["join(sorted((first+last)))]/self.freAllTran["join(sorted(first))"]

def algorithm(self):                                #Apriori 算法
    self.scanFirDatas()
    while self.traCount != {}:
        self.getFreset()
        self.selfConn()
        self.scanDatas()
        #print(self.freAllTran)
        #print(self.freTran)
        self.genAssRule()
        for i in self.rules:
            print(i)

#traDatas=['abc','ae','abc','ade']                  #使用的数据集，用于挖掘关联规则

f = open("G:\\大学\\数据挖掘\\实验 2\\lens.txt",encoding='utf-8')
data=[]
while True:                                         #将 lens.txt 文件读入，每行作为一个字符串
    line = f.readline()
    if line:
        index=line.find('\n')
        listline=list(line)
        del(listline[index:index+2])

```

```

        data.append("join(listline))
else:
    break
f.close()
for i in range(len(data)):          #将每行的字符串用“，”隔开，识别不同属性
    data[i]=data[i].split(sep=',')
variable=[]
for i in data:                      #将所有出现的属性存在列表中
    for j in i:
        if j not in variable:
            variable.append(j)

letter_word=dict()
word_letter=dict()
for i in range(len(variable)):      #建立字母与单词的对应关系，于是我们就能调用 Apriori
算法了
    word_letter[variable[i]]=chr(ord('a') + i)
    letter_word[chr(ord('a') + i)]=variable[i]
traDatas=[]
for i in data:
    string=""
    for j in i:
        string=string+join(word_letter[j])
    traDatas.append(join(sorted(string)))

apriori=Apriori(traDatas,2.0.95,letter_word)
apriori.algorithm()

```

报告评分：

指导教师签字：

一、实验室名称：主楼 A2-412

二、实验项目名称：分类

三、实验学时：3

四、实验原理

4.1 KNN 算法原理

KNN 属于 lazy learning，不会对训练样本数据进行学习，其做法是：对于一个新数据，计算它与训练集中数据的距离，选择最短的 k 个作为邻居，然后预测它的类别和 k 个邻居中其所属类别最多的一致。算法伪代码如下：

- (1) 读入并保存训练数据
- (2) 预测时，计算已知类别数据集中的点与当前点之间的距离
- (3) 选取与当前距离最小的 k 个点
- (4) 计算前 k 个点所在类别的出现频率
- (5) 返回前 k 个点出现频率最高的类别作为当前点的预测分类

4.2 决策树算法原理

决策树是一种类似流程图的树结构，决策树中每个内部结点（非叶结点）表示在一个属性上的测试，每个分支代表该测试的一个输出，每个叶子结点代表类标签。给定类标号未知元组 X，在决策树上测试 X 的属性值，跟踪一条由根到叶结点的路径，该叶结点就存放着该元组的类预测。

4.2.1 决策树的构造

核心思想在于决策树越向下生长，使节点的纯度越大。

自顶向下的分治方式构造决策树

递归的通过选择相应的测试属性，通过选择最有分类能力的属性作为决策树的当前结点，由此划分样本，其中测试属性是根据某种启发信息或者是统计信息来进行选择，如信息增益

4.2.2 决策树伪代码

输入： 训练集 D

属性集 A

过程： 函数 createTree(D,A)

1: 生成结点 node

2: If D 中样本全属于同一类别 C then

3: 将 node 标记为 C 类叶结点;return

4: end if

5: If A=空集 OR D 中样本在 A 上取值相同 then

```

6:      将 node 标记为叶结点，其类标记为 D 中样本数最多的类;return
7: end if
8: 从 A 中选择最优划分属性 a
9: for a 的每一个值 a* do
10:    为 node 生成一个分支;令 D*表示 D 中在 a 上取值为 a*的样本子集;
11:    if D*为空 then
12:        将分支结点标记为叶结点，其类标记为 D 中样本最多类
13:        return;
14:    else
15:        以 createTree(D*,A - {a})为分支结点
16:    end if
17: end for
输出： 以 node 为根节点的一棵决策树

```

4.2.3 划分选择

如何选择最优划分属性，一般而言随着划分过程的进行，我们希望决策树的分支结点所包含的样本尽可能属于同一类别，即结点的“纯度”越来越高。常用的度量样本集合纯度的方法有信息增益、信息增益率、基尼指数等。

4.2.4 信息增益

假设当前样本集合 D 中第 k 类样本所占比例为 p_k ($k=1,2,\dots,n$)，则 D 的信息熵定义为：

$$Ent(D) = - \sum_{k=1}^n p_k * \log_2 p_k$$

信息增益反映了使用属性 a 对样本集 D 进行划分后，所得到的“纯度”提升。假设离散属性 a 有 V 个可能的取值，信息增益定义为：

$$Gain(D, a) = Ent(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} Ent(D^v)$$

五、实验目的

- 1、了解分类的基本概念、原理和一般方法
- 2、掌握分类的基本算法
- 3、实现 KNN 或决策树算法

六、实验内容

- 1、实现 KNN 算法
- 2、实现决策树算法

3、（选做）实现朴素贝叶斯算法/ANN 等

七、实验器材（设备、元器件）

台式机、笔记本（MagicBook）

八、实验步骤

8.1 实现 KNN 算法

- (1) 根据实验原理，编写 KNN 算法。
- (2) 读入训练集 iris.2D.train.arff，并保存在 List 中。提供了 readARFF(File file)函数读取数据
- (3) 读入测试集 iris.2D.test.arff，调用 predict 函数进行分类，观察结果，其中测试集中的类标签用于计算分类准确率
- (4) 将实验结果截图，连同代码和注释放入实验报告

8.2 实现决策树算法

如下给出构建决策树所需实现的函数与数据结构的一种思路：

ArrayList<String> attribute

存储原始属性的名称

ArrayList<String[]> dataSet

存储原始训练数据

HashMap<String, HashMap<String, Object>> tree

保存树结构，其中 key 保存父节点，value 保存子树。其中 Object 类是 Java 中的泛型，用来实现树的递归定义

ArrayList<String[]> splitDataSet(ArrayList<String[]> data, int feature, String value)

输入：数据集合，String[]数组中保存的是各维特征的取值，其中最后一维是数据的类标签

输出：该数据集的信息熵

ArrayList<String[]> splitDataSet(ArrayList<String[]> data, int feature, String value)

输入：需分裂的数据集、用来分裂数据集的特征（用该特征在特征集合中的序号表示），特征值
输出：数据集子集，包含 feature 维特征取值为 value 的数据，并且该数据子集删去了维度 feature

int chooseBestFeatureToSplit(ArrayList<String[]> data)

输入：数据集合

输出：对 data 的每个特征维度都计算信息增益，其中会用到 splitDataSet 函数划分得到数据子集，返回信息增益最大的划分特征（用该特征在特征集合中的序号表示）

九、实验数据及结果分析

该部分内容较多，插入目录便于浏览。

目录

9.1 KNN 算法	44
9.1.1 使用 WEKA 图形界面实现 KNN	44
9.1.2 自己动手实现 KNN 算法.....	48
9.1.2.1 KNN 算法原理	48
9.1.2.2 程序设计	49
9.1.2.3 结果展示	50
9.2 人工神经网络.....	51
9.2.1 感知机	51
9.2.2 BP 网络	53
9.3 决策树分类.....	55
9.3.1 使用 WEKA 处理决策树分类问题.....	55
9.3.2 自己编程实现决策树算法.....	58
9.3.2.1 程序设计	58
9.3.2.2 结果展示	61

9.1 KNN 算法

9.1.1 使用 WEKA 图形界面实现 KNN

打开 Weka，打开文件 iris.2D.train.arff。iris.2D.train.arff 数据如图所示：

The screenshot shows a Notepad++ window with two tabs: "iris.2D.test.arff" and "iris.2D.train.arff". The "iris.2D.train.arff" tab is active, displaying the following ARFF file content:

```
1 @relation iris-weka.filters.unsupervised.attribute.Remove-R1-2
2
3 @attribute petallength numeric
4 @attribute petalwidth numeric
5 @attribute class {Iris-setosa,Iris-versicolor,Iris-virginica}
6
7 @data
8 1.6,0.2,Iris-setosa
9 1.6,0.4,Iris-setosa
10 1.5,0.2,Iris-setosa
11 1.4,0.2,Iris-setosa
12 1.6,0.2,Iris-setosa
13 1.6,0.2,Iris-setosa
14 1.5,0.4,Iris-setosa
15 1.5,0.1,Iris-setosa
16 1.4,0.2,Iris-setosa
17 1.5,0.1,Iris-setosa
18 1.2,0.2,Iris-setosa
19 1.3,0.2,Iris-setosa
20 1.5,0.1,Iris-setosa
21 1.3,0.2,Iris-setosa
22 1.5,0.2,Iris-setosa
23 1.3,0.3,Iris-setosa
24 1.3,0.3,Iris-setosa
25 1.3,0.2,Iris-setosa
26 1.6,0.6,Iris-setosa
27 1 9 0 4 Iris-setosa
```

At the bottom of the window, status bars show "Normal text file", "length: 1,929 lines: 83", "Ln: 1 Col: 1 Pos: 1", "Windows (CR LF)", "UTF-8", and "INS".

图 9.1.1 数据集一览

Weka 选项卡中选择 Classify, classifiers 选择 Lazy-IBK, 参数设置如下:

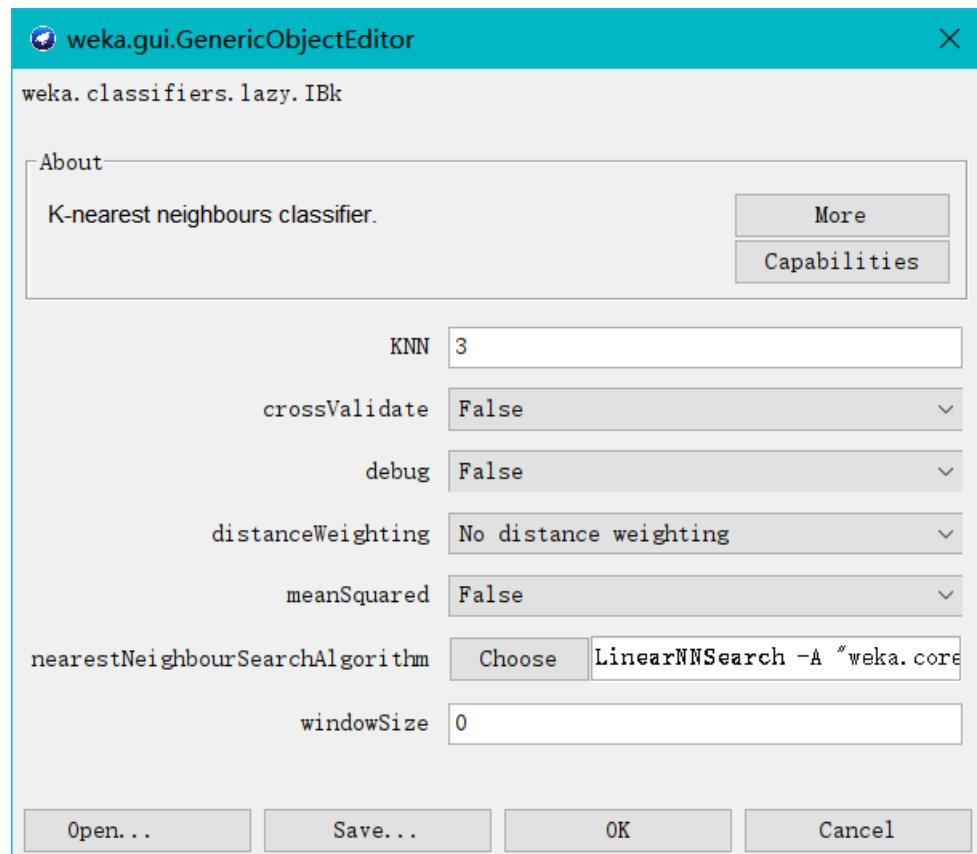


图 9.1.2 Weka 实现 KNN 的参数设置

(可以看到 k 值为 3)

设置完参数后选择 Cross-validation (交叉验证) , 点击 start, 运行结果如图所示:

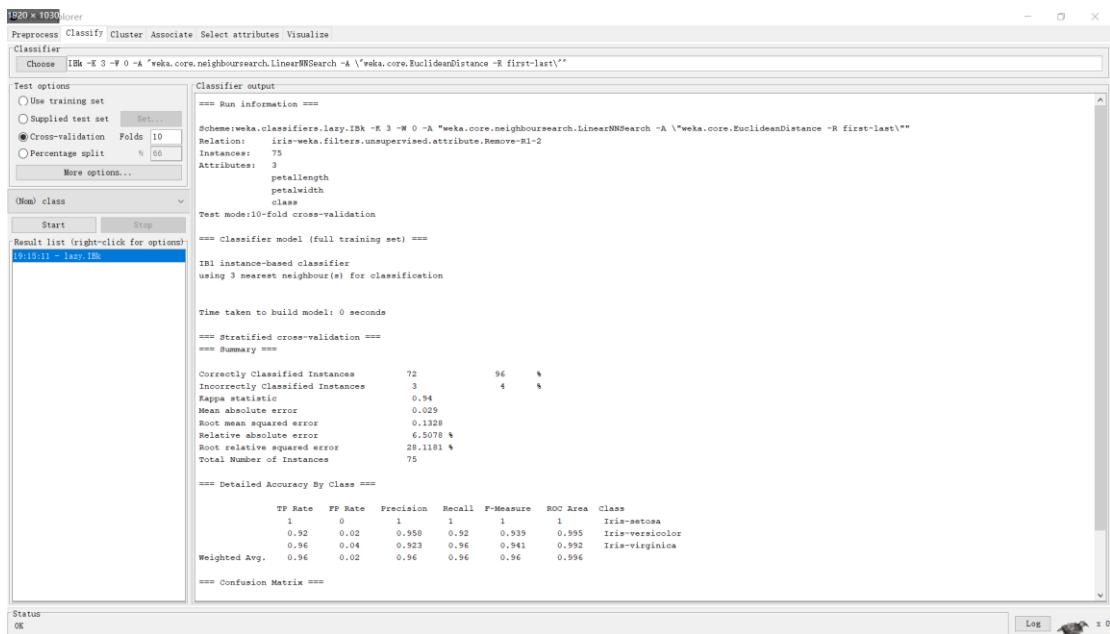


图 9.1.3 Weka 分类时的运行结果 (train)

Confusion Matrix:

a	b	c	<- classified as
25	0	0	a = Iris-setosa
0	23	2	b = Iris-versicolor
0	1	24	c = Iris-virginica

可见在 75 个数据中只有 3 个数据被分类错误。

接下来对测试文件进行测试, Test options 选择 Supplied test set, 并打开 iris.2D.test.arff 测试文件。iris.2D.test.arff 数据如图所示:

```

G:\大学\数据挖掘\实验3\数据\forKNN\iris.2D.test.arff - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?
iris.2D.test.arff iris.2D.train.arff
1 @relation iris-weka.filters.unsupervised.attribute.Remove-R1-2
2
3 @attribute petallength numeric
4 @attribute petalwidth numeric
5 @attribute class {Iris-setosa,Iris-versicolor,Iris-virginica}
6
7 @data
8 1.4,0.2,Iris-setosa
9 1.4,0.2,Iris-setosa
10 1.3,0.2,Iris-setosa
11 1.5,0.2,Iris-setosa
12 1.4,0.2,Iris-setosa
13 1.7,0.4,Iris-setosa
14 1.4,0.3,Iris-setosa
15 1.5,0.2,Iris-setosa
16 1.4,0.2,Iris-setosa
17 1.5,0.1,Iris-setosa
18 1.5,0.2,Iris-setosa
19 1.6,0.2,Iris-setosa
20 1.4,0.1,Iris-setosa
21 1.1,0.1,Iris-setosa
22 1.2,0.2,Iris-setosa
23 1.5,0.4,Iris-setosa
24 1.3,0.4,Iris-setosa
25 1.4,0.3,Iris-setosa
26 1.7,0.3,Iris-setosa
27 1.5,0.3,Iris-setosa

```

Normal text file length: 1,921 lines : 83 Ln:7 Col:6 Pos:200 Windows (CR LF) UTF-8 INS

图 9.1.4 数据集一览

右键选择 Re-evaluate Model on current test set, 用当前模型测试数据, 运行结果如下:

1920 x 1030 More

Preprocess Classify Cluster Associate Select attributes Visualize

Classifier Choose: IBk - K 3 - W 0 -A "weka.core.neighboursearch.LinearNNSearch -A \"weka.core.EuclideanDistance -R first-last\""

Test options

- Use training set
- Supplied test set Set...
- Cross-validation Folds 10
- Percentage split % 00
- More options...

(Nom) class

Start Stop

Result list (right-click for options): 19:15:11 - lazy.IBk 19:45:10 ~ lazy.IBk

Classifier output

```

==== Run information ====
Scheme:weka.classifiers.lazy.IBk -K 3 -W 0 -A "weka.core.neighboursearch.LinearNNSearch -A \"weka.core.EuclideanDistance -R first-last\""
Relation: iris=weka.filters.unsupervised.attribute.Remove-R1-2
Instances: 75
Attributes: 3
petallength
petalwidth
class
Test mode:user supplied test set: size unknown (reading incrementally)
==== Classifier model (full training set) ====
IBk instance-based classifier
using 3 nearest neighbour(s) for classification

Time taken to build model: 0 seconds

==== Evaluation on test set ====
==== Summary ====
Correctly Classified Instances 73 95.3333 %
Incorrectly Classified Instances 2 2.6667 %
Kappa statistic 0.96
Mean absolute error 0.029
Root mean squared error 0.1254
Relative absolute error 6.5225 %
Root relative squared error 26.6077 %
Total Number of Instances 75

==== Detailed Accuracy By Class ====

```

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
1	0	1	1	1	1	1	Iris-setosa
0.96	0.02	0.96	0.96	0.96	0.963	0.963	Iris-versicolor
0.96	0.02	0.96	0.96	0.96	0.969	0.969	Iris-virginica
Weighted Avg.	0.973	0.013	0.973	0.973	0.973	0.991	

```

==== Confusion Matrix ====

```

图 9.1.5 Weka 分类时的运行结果 (test)

Confusion Matrix:

a	b	c	<- classified as
25	0	0	a = Iris-setosa
0	24	1	b = Iris-versicolor
0	1	24	c = Iris-virginica

可见在 75 个数据中只有 2 个数据被分类错误。

下图为分类错误的可视化：

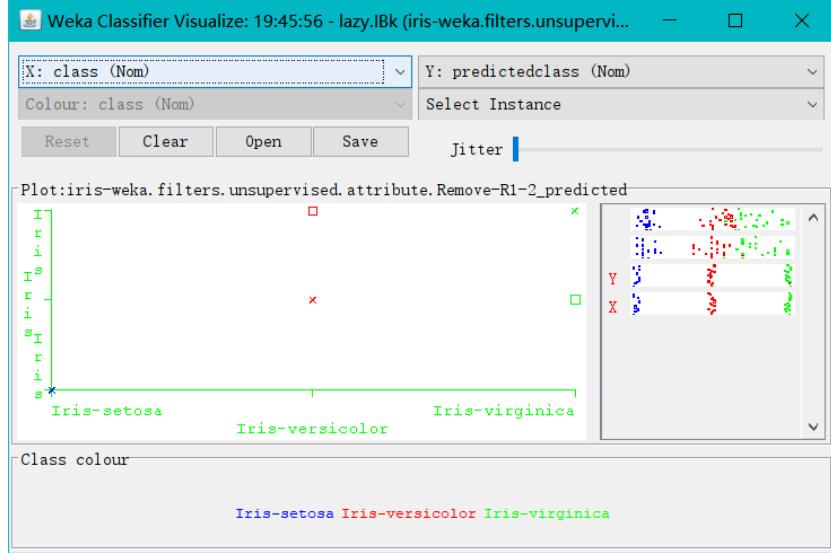


图 9.1.5 Visualize classifier errors

将该错误可视化文件保存为 arff 格式，打开如图所示：

The figure shows a Notepad++ window displaying the 'test_result.arff' file. The file contains 75 data points, each consisting of a numerical ID followed by a class label. The 'Iris-virginica' class is highlighted with a red rectangle around the entry for ID 48. The file is a normal text file with 2,994 length and 83 lines. The encoding is set to Unix (LF) and UTF-8.

ID	Class
49	Iris-versicolor
50	Iris-versicolor
51	Iris-versicolor
52	Iris-versicolor
53	Iris-versicolor
54	Iris-virginica
55	Iris-versicolor
56	Iris-versicolor
57	Iris-versicolor
58	Iris-versicolor
59	Iris-virginica
60	Iris-virginica
61	Iris-virginica
62	Iris-virginica
63	Iris-virginica
64	Iris-virginica
65	Iris-virginica
66	Iris-virginica
67	Iris-virginica
68	Iris-virginica
69	Iris-virginica
70	Iris-virginica
71	Iris-virginica
72	Iris-virginica
73	Iris-virginica
74	Iris-virginica
75	Iris-virginica

图 9.1.6 预测结果

可见只有少量的数据被预测错误，KNN 在此数据集下的分类结果极佳。

9.1.2 自己动手实现 KNN 算法

9.1.2.1 KNN 算法原理

KNN 属于 lazy learning，不会对训练样本数据进行学习，其做法是：对于一个新数据，

计算它与训练集中数据的距离，选择最短的 k 个作为邻居，然后预测它的类别和 k 个邻居中其所属类别最多的一致。

KNN 算法优点：简单，快速，准确率较高

KNN 算法缺点：

(1) K 值的选择

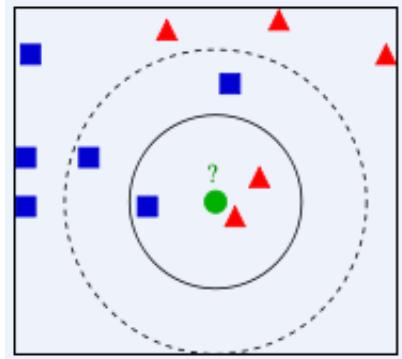


图 9.1.7 不同 k 值导致分类矛盾

(2) 训练数据

由于 KNN 属于 lazy learning，不会对训练数据集做深入学习，而是单纯的考虑数据之间的距离，所以 KNN 的预测效果强依赖于训练数据。KNN 的不足体现在：

A. 训练样本能否正确体现数据真实分布

比如，训练样本是否大多属于某一个类，或某几个类；

B. 训练数据是否属于同一分布

比如，对噪声数据敏感

(3) 其他

比如距离的选定，距离的计算复杂性

三个步骤：(1) 算距离 (2) 找邻居 (3) 投票分类

9.1.2.2 程序设计

结合以上实验原理，可实现 KNN 的 Python 程序设计。

首先打开训练数据文件，并将数据读入 dataset 中：

```
with open(r'G:\大学\数据挖掘\实验3\数据\forKNN\iris.2D.train.txt', 'r') as fin:      #打开训练集文件
    for line in fin.readlines():
        data=(line.strip().split(','))
        dataset.append(data)
```

图 9.1.8 读入训练数据的代码

再打开测试数据文件，并将数据读入 testdataset 中：

```
with open(r'G:\大学\数据挖掘\实验3\数据\forKNN\iris.2D.test.txt', 'r') as fin:      #打开测试集文件
    for line in fin.readlines():
        data=(line.strip().split(','))
        testdataset.append(data)
```

图 9.1.9 读入测试数据的代码

其次是核心函数 findNN、vote 函数，分布用于找到原数据中距测试数据最近的 k 个值并返回、返回 k 个最邻近数据中相同标签数最多的值（统计距离测试点最近的 k 的训练数据的标签，出现次数最多的标签就是测试数据的预测标签）。其中调用了求欧氏距离的函数 computeDis。

```

def findNN(dataset, testData, k): ..... #找到原数据中距测试数据最近的k个值并返回
    distances=[]
    i=0
    for data in dataset:
        distances.append([computeDis(testData,data),i])
        i+=1
    distances_s=sorted(distances, key=lambda m: m[0])[:k] ..... #距离最小的k个数据
    id=[distances_s[i][1] for i in range(len(distances_s))] ..... #距离最小的k个数据的序号
    return id

def computeDis(x,y): ..... #计算欧氏距离
    return math.sqrt(math.pow(float(x[0])-float(y[0]),2)+math.pow(float(x[1])-float(y[1]),2))

def vote(dataset, indexs, votes): ..... #返回k个最近邻数据中相同标签数最多的值
    for i in indexs:
        label=dataset[i][2]
        if label not in votes.keys():
            votes[label]=1
        else:
            votes[label]+=1
    return (max(votes, key=votes.get))

```

图 9.1.10 findNN、computeDis 及 vote 函数代码

main 函数能够完成读入测试数据，预测新数据的分类并输出的功能。同时计算了分类准确率。

```

for i in range(len(testdataset)):
    votes={}
    indexs=findNN(dataset,[testdataset[i][0],testdataset[i][1]],3)
    vote_result.append(vote(dataset,indexs,votes))

for i in range(len(testdataset)):
    print('data: '+testdataset[i][0]+' '+testdataset[i][1]) ..... #计算得到每个测试数据的分类
    print('real_label: '+testdataset[i][2]+'\nestimated_label: '+vote_result[i]+'\n') ..... #将数据打印处理
    all_right_num=sum([vote_result[i]==testdataset[i][2] for i in range(len(testdataset))])
    print('RightRate: '+str(all_right_num/len(testdataset)*100)+'%') ..... #将数据真实标签和预测标签作比较
    #计算分类正确的百分比

```

图 9.1.11 findNN、computeDis 及 vote 函数代码

全部源码详见附录。

9.1.2.3 结果展示

```

1920 x 1030 100%  实验3 - KNN.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help 实验3 - KNN.py
Project KNNDemo KNNDemo testpy
Run: KNNDemo
data:1.4 0.2
real_label:Iris-setosa
estimated_label:Iris-setosa
data:1.3 0.2
real_label:Iris-setosa
estimated_label:Iris-setosa
data:1.5 0.2
real_label:Iris-setosa
estimated_label:Iris-setosa
data:1.4 0.2
real_label:Iris-setosa
estimated_label:Iris-setosa
data:1.7 0.4
real_label:Iris-setosa
estimated_label:Iris-setosa
data:1.4 0.3
real_label:Iris-setosa
estimated_label:Iris-setosa
data:1.5 0.2
real_label:Iris-setosa
estimated_label:Iris-setosa
data:1.4 0.2
real_label:Iris-setosa
estimated_label:Iris-setosa
data:1.5 0.1
real_label:Iris-setosa
estimated_label:Iris-setosa
Run TODO Problems Terminal Python Console Event Log
7:23 CRLF UTF-8 4 spaces Python 3.8
Type In word: Rightrate

```



```

File Edit View Navigate Code Refactor Run Tools VCS Window Help 实验3 - KNN.py
Project KNNDemo KNNDemo testpy
Run: KNNDemo
data:5.9 2.3
real_label:Iris-virginica
estimated_label:Iris-virginica
data:5.1 1.5
real_label:Iris-virginica
estimated_label:Iris-versicolor
data:5.7 2.3
real_label:Iris-virginica
estimated_label:Iris-virginica
data:4.9 2
real_label:Iris-virginica
estimated_label:Iris-virginica
data:6.7 2
real_label:Iris-virginica
estimated_label:Iris-virginica
data:4.9 1.8
real_label:Iris-virginica
estimated_label:Iris-virginica
data:5.7 2.1
real_label:Iris-virginica
estimated_label:Iris-virginica
Rightrate: 96.0%
Process finished with exit code 0
Run TODO Problems Terminal Python Console Event Log
296:31 CRLF UTF-8 4 spaces Python 3.8
Type In word: Rightrate

```

图 9.1.12 部分输出结果

可见大部分数据都分类正确，正确率 96.0%。与 weka 的 $(75-2) / 75 = 97.3\%$ 十分接近，可见程序运行结果符合预期。

9.2 尝试利用 matlab 实现人工神经网络，包括感知机和后向传播网络（BP）

9.2.1 感知机

写 matlab 代码如图，实现感知机。

```

clear all
data=[3 3 1;
      4 3 1;
      1.5 0 1;
      0.5 0.9 1;
      2 1 1;
      0.5 0.5 -1;
      0 0 -1;
      -2 -2 -1;
      -3 0 -1;
      1 -1 -1];
X=data(:,[1,2]); y=data(:,3);
m=size(X,1); % m=样本点个数
plotData2(X,y);%先在图上将样本画出来
axis([-4 4 -4 4]);
hold on
x1=-4:0.2:4; %x1坐标轴

```

图 9.2.1 数据输入及图形绘制

```

function plotData2(X,y)
figure;hold on;
pos=find(y==1);
neg=find(y==-1);
plot(X(pos,1),X(pos,2),'k+', 'Linewidth',2,'MarkerSize',9);
plot(X(neg,1),X(neg,2),'ko','MarkerFaceColor','r','Linewidth',2,'MarkerSize',7);
hold off;
end

```

图 9.2.2 绘图函数

```

W=[0;0]; b=0;
alph=0.1; %学习率
error=1;
while error>0
    error=0;
    for i=1:m
        if (((W'*X(i,:)+b)*y(i))<=0)
            error=error+1;
            W=W+alph*y(i)*X(i,:);
            b=b+alph*y(i);
            y1=(-W(1)*x1-b)/W(2);
            plot(x1,y1,'-b');
            pause(1);
        end
    end
    plot(x1,y1,'-r','Linewidth',3);

```

图 9.2.3 感知机核心代码

点击运行可以观察到运行结果：

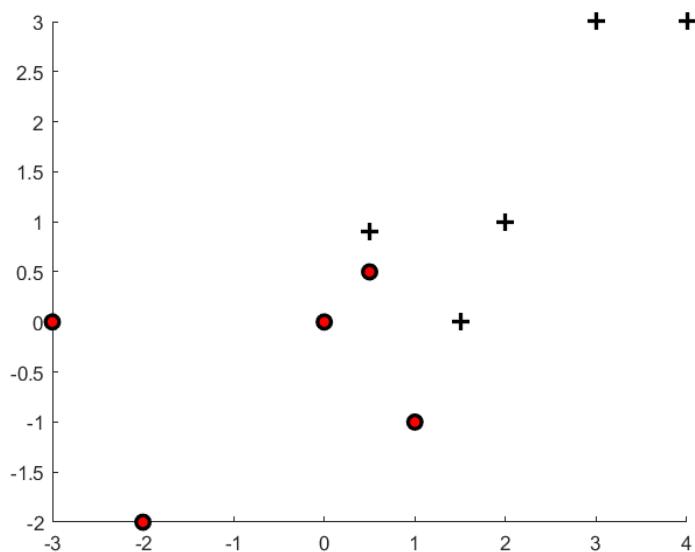


图 9.2.4 输入向量及对应标签

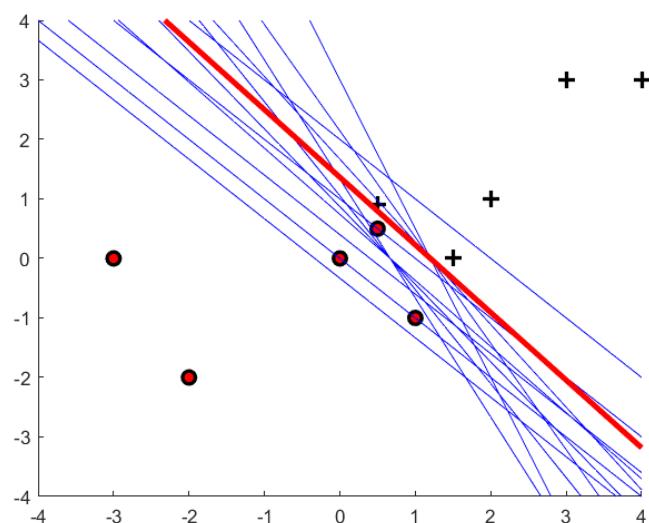


图 9.2.5 多次分类结果和最终结果

如图，蓝线为每次迭代得到的划分结果，红线为最终划分结果。可见，经过多次迭代，得到了较为满意的“超平面”，完成感知机功能。

9.2.2 BP 网络

写 matlab 代码如图，实现 BP 网络。

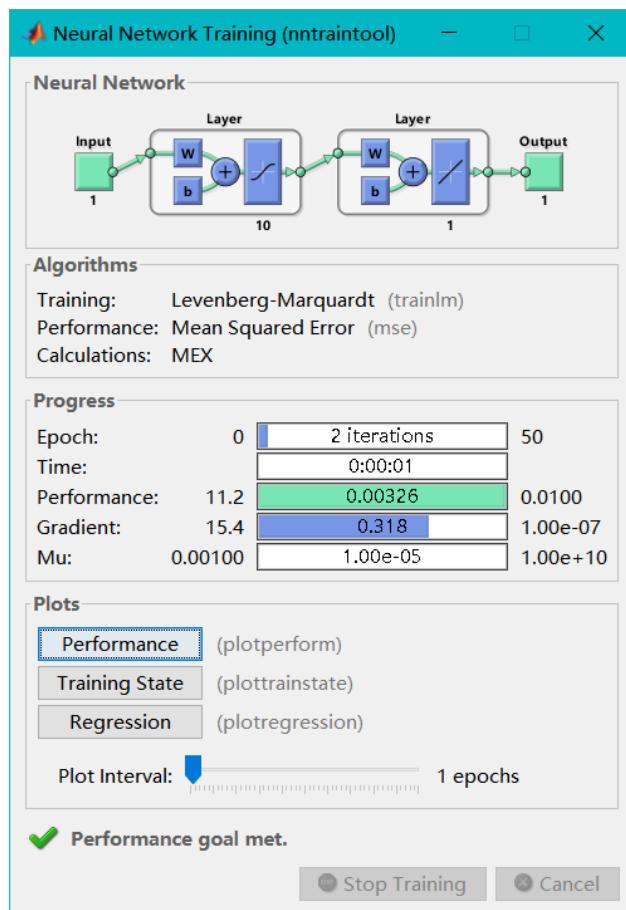


图 9.2.3 参数设置

设置了 10 层的神经网络。

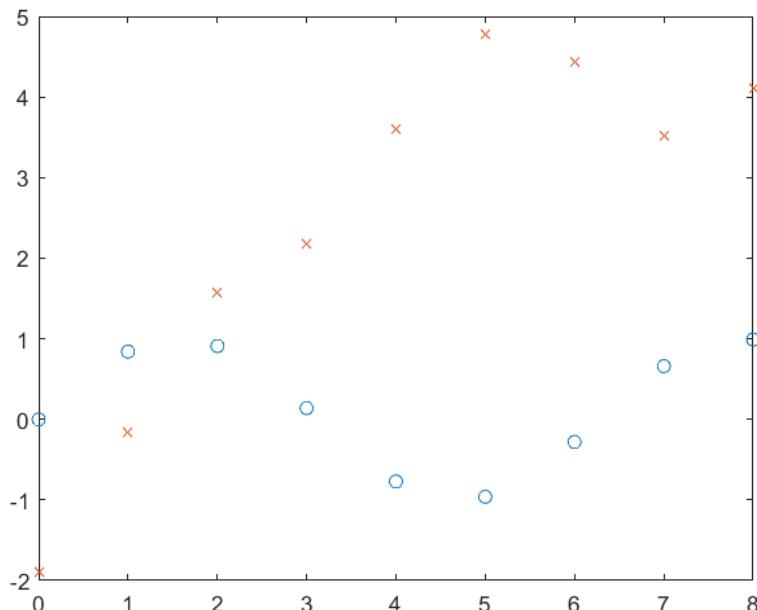


图 9.2.4 初始网络预测结果

可见初始的网络对数据的预测完全不准确。

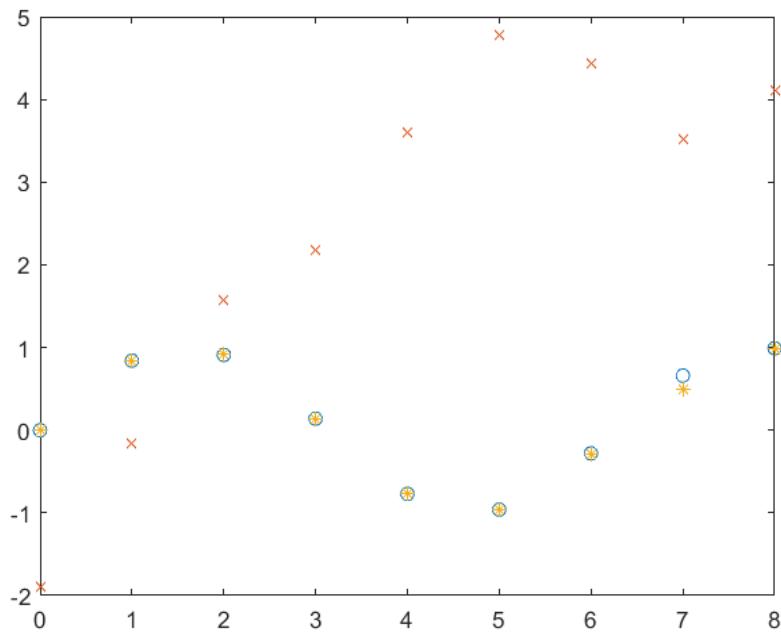


图 9.2.5 训练后网络预测结果

训练后的网络预测结果用圆圈表示，可见预测结果很好，只有少量点有细微偏差。

9.3 决策树分类

9.3.1 使用 WEKA 处理决策树分类问题

导入数据集 dataset.arff, dataset.arff 数据如图所示：

```

G:\大学\数据挖掘\实验3\数据\forDecisionTree\dataset.arff - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?
0 实验课1 题目1.cpp 1 iris.2D.test.arff 2 iris.2D.train.arff 3 iris.arff 4 dataset.arff
4 @attribute havehouse numeric
5 @attribute credit numeric
6 @attribute class {'no','yes'}
7 @data
8 0,0,0,0,'no'
9 0,0,0,1,'no'
10 0,1,0,1,'yes'
11 0,1,1,0,'yes'
12 0,0,0,0,'no'
13 1,0,0,0,'no'
14 1,0,0,1,'no'
15 1,1,1,1,'yes'
16 1,0,1,2,'yes'
17 1,0,1,2,'yes'
18 2,0,1,2,'yes'
19 2,0,1,1,'yes'
20 2,1,0,1,'yes'
21 2,1,0,2,'yes'
22 2,0,0,0,'no'
23

```

Normal text file | length : 383 | lines : 23 | Ln : 1 | Col : 1 | Pos : 1 | Windows (CR LF) | UTF-8 | INS | |

图 9.3.1 数据集一览

选择 Classify, classifier 选择 tree 中的 j48, 也即决策树, 设置参数如图:

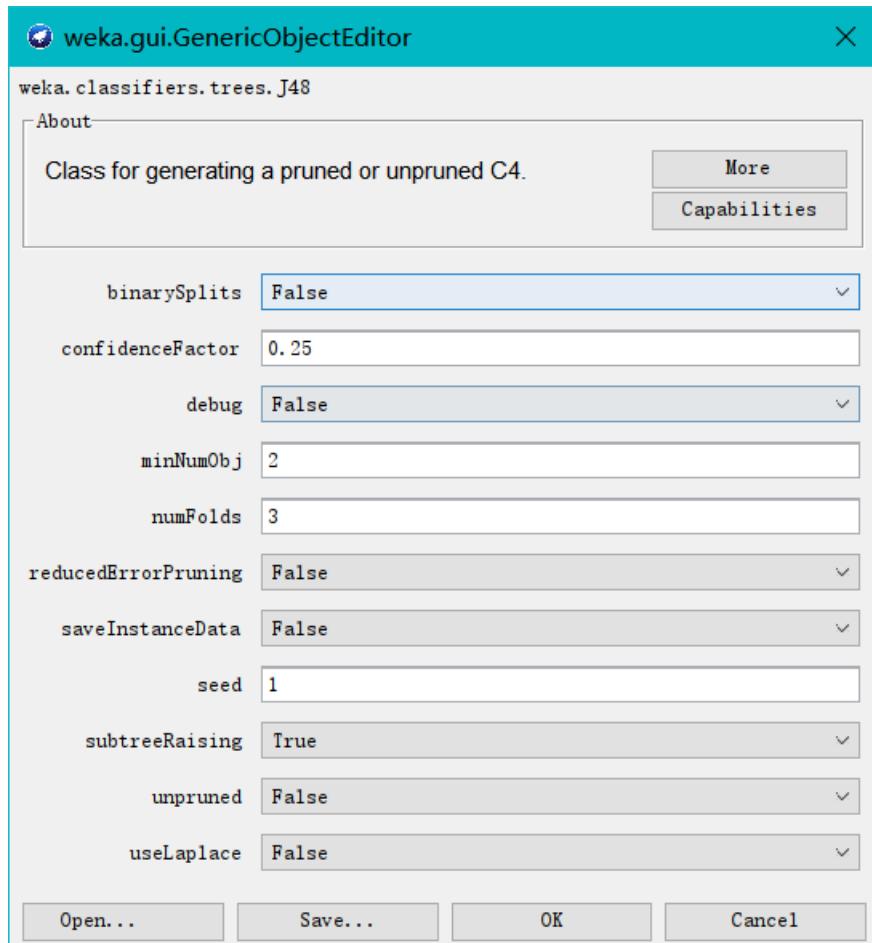


图 9.3.2 参数设置

点击运行, 运行结果如图所示。

```

1920 x 1030 J48
Preprocess Classify Cluster Associate Select attributes Visualize
Classifier: Choose J48 -C 0.25 -M 2
Test options:
  (radio) Use training set
  (radio) Supplied test set Set...
  (radio) Cross-validation Folds 10 %
  (radio) Percentage split % 66
  More options...
  (button) Class
  Start Stop
Result list (right-click for options): 11 01 4 - trees.J48
  Test mode: 10-fold cross-validation
  Classifier model (full training set) ===
  J48 pruned tree
  -----
  havehouse <= 0
  | havework <= 0: no (6.0)
  | havework > 0: yes (3.0)
  havehouse > 0: yes (4.0)

  Number of Leaves : 3
  Size of the tree : 5

  Time taken to build model: 0.01 seconds

  === Stratified cross-validation ===
  === Summary ===

  Correctly Classified Instances      15      100      %
  Incorrectly Classified Instances   0       0      %
  Kappa statistic                   1
  Mean absolute error               0
  Root mean squared error           0
  Relative absolute error           0      %
  Root relative squared error      0      %
  Total Number of Instances        15

  === Detailed Accuracy By Class ===

    TP Rate   FP Rate   Precision   Recall   F-Measure   ROC Area   Class
    1         0         1         1         1         1         no
    1         0         1         1         1         1         yes
  Weighted Avg. 1         0         1         1         1         1

  === Confusion Matrix ===

  a b  <-- classified as
  6 0  | a = no
  0 9  | b = yes

```

图 9.3.3 决策树运行结果

分类结果如下：

Confusion Matrix:

a	b	<-- classified as
6	0	a = no
0	9	b = yes

选择 Visualize Tree 以可视化决策树：

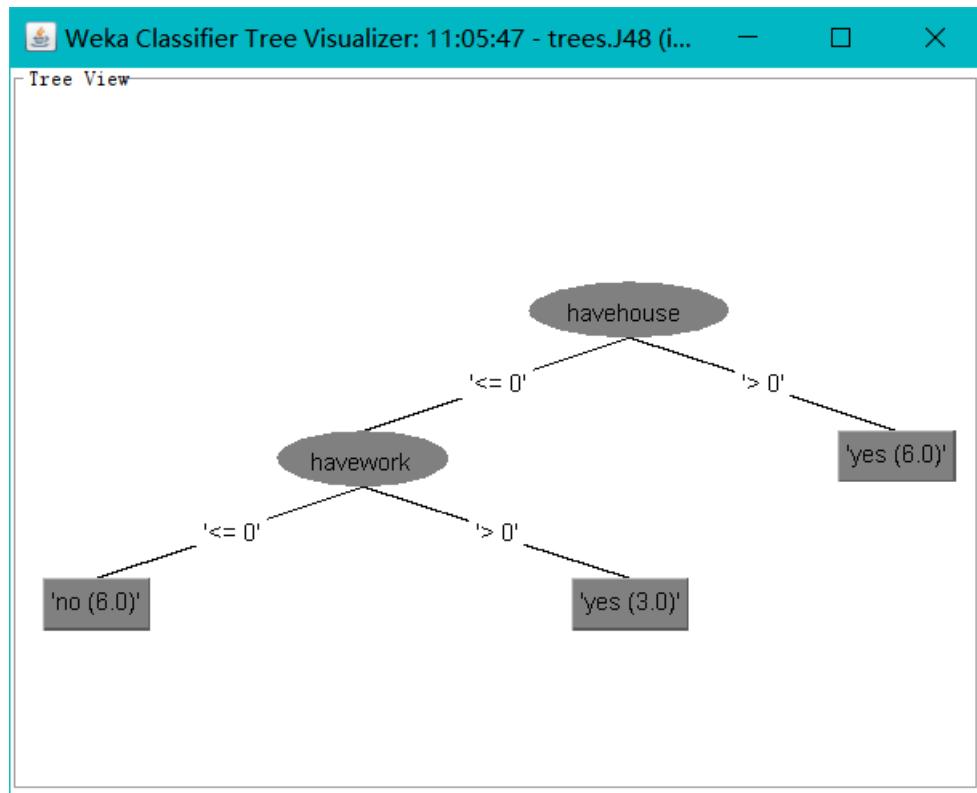


图 9.3.4 决策树

9.3.2 自己编程实现决策树算法

9.3.2.1 程序设计

设计多个决策树函数以及主程序。

数据初始化函数 `createDataSet`, 用于产生提供所需数据。

```

def createDataSet(): #创建数据集
    dataSet = [[0,0,0,0, 'no'],
               [0,0,0,1, 'no'],
               [0,0,0,1, 'yes'],
               [0,1,0,0, 'yes'],
               [0,1,0,0, 'no'],
               [1,0,0,0, 'no'],
               [1,0,0,1, 'no'],
               [1,1,1,1, 'yes'],
               [1,0,1,2, 'yes'],
               [2,0,1,2, 'yes'],
               [2,0,1,1, 'yes'],
               [2,1,0,1, 'yes'],
               [2,1,0,2, 'yes'],
               [2,0,0,0, 'no']]
    labels = ['年龄', '有工作否', '有房子否', '信贷情况'] #特征标签
    #change to discrete values
    return dataSet, labels
  
```

图 9.3.5 数据初始化函数

如图, 数据含年龄、有工作否、有房子否、信贷情况这四个特征标签, 最后一维表示在前四个标签下是否放贷。

信息熵相关函数有 `calcShannonEnt` 和 `chooseBestFeatureToSplit`。前者用于计算一个样本的信息熵, 在计算条件信息熵时也会使用到。后者用于选择哪个特征对应信息增益最大。

```

##### 计算信息熵 #####
def calcShannonEnt(dataSet):
    numEntries = len(dataSet) # 样本数
    labelCounts = {} # 创建一个数据字典，key是最后一列的数值（即标签，也就是目标分类的类别），value是属于该类别的样本个数
    for featVec in dataSet:# 遍历整个数据集，每次取一行
        currentLabel = featVec[-1] # 取该行最后一列的值
        if currentLabel not in labelCounts.keys(): labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1
    shannonEnt = 0.0 # 初始化信息熵
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries
        shannonEnt -= prob * log(prob,2) #log base 2 计算信息熵
    return shannonEnt

```

图 9.3.6 信息熵计算函数 calcShannonEnt

```

##### 选取当前数据集中，用于划分数据集的最优特征 #####
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1 # 获取当前数据集的特征个数，最后一列是分类标签
    baseEntropy = calcShannonEnt(dataSet) # 计算当前数据集的信息熵
    bestInfoGain = 0.0; bestFeature = -1 # 初始化最大信息增益和最优的特征
    for i in range(numFeatures): # 遍历每个特征 iterate over all the features
        featList = [example[i] for example in dataSet] # 获取数据集中当前特征下的所有值
        uniqueVals = set(featList) # 获取当前特征值，例如年龄下有0、1、2
        newEntropy = 0.0
        for value in uniqueVals: # 计算每种划分方式的信息熵
            subDataSet = splitDataSet(dataSet, i, value)
            prob = len(subDataSet)/float(len(dataSet))
            newEntropy += prob * calcShannonEnt(subDataSet)
        infoGain = baseEntropy - newEntropy # 计算信息增益
        if (infoGain > bestInfoGain): # 比较每个特征的信息增益，只要最好的信息增益
            bestInfoGain = infoGain # if better than current best, set to best
            bestFeature = i
    return bestFeature # returns an integer

```

图 9.3.6 最大信息增益特征选择函数 chooseBestFeatureToSplit

接下来是决策树生成相关函数 createTree、splitDataSet 和 majorityCnt。createTree 使用递归，直到数据集全纯或待选特征只剩一个时停止。同时调用了数据集划分函数 splitDataSet，按给定的特征和值返回摸出该特征的数据集，用于信息增益计算； majorityCnt 函数用于找到数据集中某特征对应值最多的数据。

```

##### 按给定的特征划分数据 #####
def splitDataSet(dataSet, axis, value): # axis是dataSet数据集下要进行特征划分的列号例如outlook是0列，value是该列下某个特征值，0列中的sunny
    retDataSet = []
    for featVec in dataSet: # 遍历数据集，并抽取按axis的当前value特征进行划分的数据集(不包括axis列的值)
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis] # chop out axis used for splitting
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
    # print axis,value,reducedFeatVec
    # print retDataSet
    return retDataSet

```

图 9.3.7 数据集划分函数 splitDataSet

```

def majorityCnt(classList):
    classCount={}
    for vote in classList:
        if vote not in classCount.keys(): classCount[vote] = 0
        classCount[vote] += 1
    sortedClassCount = sorted(classCount.items(), key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]

```

图 9.3.8 majorityCnt 函数

```

##### 生成决策树主方法
def createTree(dataSet,labels,featLabels):
    classList = [example[-1] for example in dataSet] # 返回当前数据集下标签列所有值
    if classList.count(classList[0]) == len(classList):
        return classList[0] #当类别完全相同时则停止继续划分，直接返回该类的标签
    if len(dataSet[0]) == 1: #遍历完所有的特征时，仍然不能将数据集划分成仅包含唯一类别的分组 dataSet
        return majorityCnt(classList) #由于无法简单的返回唯一的类标签，这里就返回出现次数最多的类别作为返回值
    bestFeat = chooseBestFeatureToSplit(dataSet) # 获取最好的分类特征索引
    bestFeatLabel = labels[bestFeat] #获取该特征的名字
    featLabels.append((bestFeatLabel))
    # 这里直接使用字典变量来存储树信息，这对于绘制树形图很重要。
    myTree = {bestFeatLabel:{}} #当前数据集选取最好的特征存储在bestFeat中
    del(labels[bestFeat]) #删除已经在选取的特征
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value),labels,featLabels)
    return myTree

```

图 9.3.9 决策树生成函数 createTree

然后是分类预测函数 classify。通过已有决策树和使用到的特征来对新数据进行分类，返回预测的类标签。

```

def classify(inputTree,featLabels,testVec): #通过已有决策树和使用到的特征来对新数据进行分类，返回类标签
    firstStr=next(iter(inputTree))
    secondDict = inputTree[firstStr]
    featIndex = featLabels.index(firstStr)+1
    for key in secondDict.keys():
        if testVec[featIndex]==key:
            if type(secondDict[key]).__name__=='dict':
                classLabel=classify(secondDict[key],featLabels,testVec)
            else:
                classLabel=secondDict[key]
    return classLabel

```

图 9.3.10 分类预测函数 classify

最后是主函数 main，用于测试决策树实验的结果。

```

if __name__ == '__main__':
    dataSet_labels=createDataSet()
    featLabels=[]
    myTree=createTree(dataSet_labels,featLabels)      #生成决策树
    #测试决策树的分类效果
    dataSet = [[0, 0, 0, 0, 'no'],
               [0, 0, 0, 1, 'no'],
               [0, 1, 0, 1, 'yes'],
               [0, 1, 1, 0, 'yes'],
               [0, 0, 0, 0, 'no'],
               [1, 0, 0, 0, 'no'],
               [1, 0, 0, 1, 'no'],
               [1, 1, 1, 1, 'yes'],
               [1, 0, 1, 2, 'yes'],
               [2, 0, 1, 2, 'yes'],
               [2, 0, 1, 1, 'yes'],
               [2, 1, 0, 1, 'yes'],
               [2, 1, 0, 2, 'yes'],
               [2, 0, 0, 0, 'no']]
    i=0
    rightnum=0
    for data in dataSet:
        i+=1
        testVec=data[0:4]
        result=classify(myTree,featLabels,testVec)
        print('数据及标签: ')
        print(data)
        print('预测标签: '+result+'\n')           #原始数据、预测数据对比
        if(result==data[-1]):
            rightnum+=1
        print('决策树: ')
        print(myTree)                           #显示决策树
        print('\n')
    print('rightratio: '+str(rightnum/len(dataSet)*100)+'%') #显示分类正确率

```

图 9.3.11 主函数 main

9.3.2.2 结果展示

部分运行结果如图所示。

```

0:00 * 1024 Python View Navigator Code Refactor Run Tools VCS Window Help  实例3 - decision_tree2.py
实例3  P  X  Decision tree2.py
Project  Run:  Decision tree2
0:00 * 1024 Python3.6.8\python.exe G:/大学/数据挖掘/实验3/decision_tree2.py
Decision tree2.py
数据及标签:
[0, 0, 0, 0, 'no']
预测标签: no
数据及标签:
[0, 0, 0, 1, 'no']
预测标签: no
数据及标签:
[0, 1, 0, 1, 'yes']
预测标签: yes
数据及标签:
[0, 1, 1, 0, 'yes']
预测标签: yes
数据及标签:
[0, 0, 0, 0, 'no']
预测标签: no
数据及标签:
[1, 0, 0, 0, 'no']
预测标签: no
数据及标签:
[1, 0, 0, 1, 'no']
预测标签: yes
数据及标签:
[1, 1, 1, 1, 'yes']
预测标签: yes
数据及标签:
[1, 0, 1, 2, 'yes']
预测标签: yes
数据及标签:
[2, 0, 1, 2, 'yes']
预测标签: yes
数据及标签:
[2, 0, 1, 1, 'yes']
预测标签: yes
数据及标签:
[2, 1, 0, 1, 'yes']
预测标签: yes
数据及标签:
[2, 1, 0, 2, 'yes']
预测标签: yes
数据及标签:
[2, 0, 0, 0, 'no']

rightratio: 100%

```

图 9.3.12 部分运行结果

```
决策树:  
{'有工作否': {0: {'有房子否': {0: 'no', 1: 'yes'}}, 1: 'yes'}}  
  
rightratio: 100.0%
```

图 9.3.13 决策树及正确率展示

```
yes  
  
Process finished with exit code 0
```

图 9.3.14 针对测试数据[1,1,0,0]得到的分类结果

可见对所有数据我们都完成测试，正确率 100.0%。决策树采用 set 集合形式组织。

针对测试数据[1,1,0,0]得到的分类结果为“yes”。

十、实验结论

成功利用 Weka 完成了 KNN 算法实践、决策树生成展示及预测，以及通过 matlab 代码实现感知机、BP 神经网络，用 Python 自己动手实现了 KNN 算法和决策树算法。

针对特定数据集，KNN 和决策树可以有较高准确率。人工神经网络使用 matlab，代码将变得十分简单。

十一、总结及心得体会

成功利用 Weka 完成了 KNN 算法实践、决策树生成展示及预测，以及通过 matlab 代码实现感知机、BP 神经网络，用 Python 自己动手实现了 KNN 算法和决策树算法。使用 Weka 实现和自己动手实践得到了相似的分类结果，符合实验预期。通过这次实验我对数据的认识更加深刻，提高了动手能力，也更加理解了 KNN 算法、决策树算法、人工神经网络，并对数据挖掘有了更浓厚的兴趣，希望以后能多进行类似实验。

十二、对本实验过程及方法、手段的改进建议

建议提前发出 PPT 及指导书，以便学生预习。

提供多种多样的数据集，以便测试。

十三、源码

自己动手实现 KNN 算法：

```
import math

def findNN(dataset,testData,k):          #找到原数据中距测试数据最近的 k 个
    值并返回
    distances=[]
    i=0
    for data in dataset:
        distances.append([computeDis(testData,data),i])
        i+=1
    distances_s=sorted(distances,key=lambda m: m[0])[:k]      #距离最小的 k 个数据
    id=[distances_s[i][1] for i in range(len(distances_s))]  #距离最小的 k 个数据的序号
    return id

def computeDis(x,y):                     #计算欧氏距离
    return
    math.sqrt(math.pow(float(x[0])-float(y[0]),2)+math.pow(float(x[1])-float(y[1]),2))

def vote(dataset,indexs,votes):          #返回 k 个最邻近数
    据中相同标签数最多的值
    for i in indexs:
        label=dataset[i][2]
        if label not in votes.keys():
            votes[label]=1
        else:
            votes[label]+=1
    return (max(votes,key=votes.get))

if __name__=='__main__':
    dataset = []
    distances = []
    testdataset=[]

    with open(r'G:\大学\数据挖掘\实验 3\数据\forKNN\iris.2D.train.txt', 'r') as fin:
        #打开训练集文件
        for line in fin.readlines():
```

```

        data=(line.strip().split(','))
        dataset.append(data)

    with open(r'G:\大学\数据挖掘\实验3\数据\forKNN\iris.2D.test.txt', 'r') as fin:
#打开测试集文件
        for line in fin.readlines():
            data=(line.strip().split(','))
            testdataset.append(data)
        vote_result=[]
        for i in range(len(testdataset)):
#计算得到每个测试数据的分类
            votes={}
            indexs=findNN(dataset,[testdataset[i][0],testdataset[i][1]],3)
            vote_result.append(vote(dataset,indexs,votes))

        for i in range(len(testdataset)):
            print('data:' + testdataset[i][0] + ' ' + testdataset[i][1])
#将数据打印处理
            print('real_label:' + testdataset[i][2] + '\nestimated_label:' + vote_result[i] + '\n') #将
数据真实标签和预测标签作比较
        all_right_num=sum([vote_result[i]==testdataset[i][2] for i in
range(len(testdataset))])
        print('Rightrate:' + str(all_right_num/len(testdataset)*100) + '%')
#计算分类正确的百分比

```

感知机

```

clear all
data=[3 3 1;
      4 3 1;
      1.5 0 1;
      0.5 0.9 1;
      2 1 1;
      0.5 0.5 -1;
      0 0 -1;
      -2 -2 -1;
      -3 0 -1;
      1 -1 -1];
X=data(:,[1,2]); y=data(:,3);
m=size(X,1); % m=样本点个数
plotData2(X,y);%先在图上将样本画出来
pause(2)
axis([-4 4 -4 4]);
hold on

```

```

x1=-4:0.2:4; %x1 坐标轴

W=[0;0]; b=0;
alph=0.1; %学习率
error=1;
while error>0
error=0;
for i=1:m
if (((W'*X(i,:)+b)*y(i))<=0)
error=error+1;
W=W+alph*y(i)*X(i,:);
b=b+alph*y(i);
y1=(-W(1)*x1-b)/W(2);
plot(x1,y1, '-b');
pause(1);
end
end
end
plot(x1,y1, '-r', 'Linewidth',3);

function plotData2(X,y)
figure;hold on;
pos=find(y==1);
neg=find(y==-1);
plot(X(pos,1),X(pos,2), 'k+', 'Linewidth',2,'MarkerSize',9);
plot(X(neg,1),X(neg,2), 'ko', 'MarkerFaceColor','r', 'Linewidth',2,'MarkerSize',7);
hold off;
end

```

BP 网络

```

clc;warning off;
p=[0 1 2 3 4 5 6 7 8];
t=[0 0.84 0.91 0.14 -0.77 -0.96 -0.28 0.66 0.99];
%建立一个前向网络, 隐层 10, 输出 1 个单元, 训练函数为 trainlm
net=newff([0 8],[10 1],{'tansig' 'purelin'},'trainlm');
y1=sim(net,p);
plot(p,t, 'o', p,y1, 'x');
%设置训练参数
net.trainParam.epochs=50;
net.trainParam.goal=0.01;
%训练网络

```

```

net=train(net,p,t);
%仿真
y2=sim(net,p);
%test
test=6.5;
y3=sim(net,test);
plot(p,t,'o',p,y1,'x',p,y2,'*');

```

自己编程实现决策树算法

```

from math import log
import operator

def createDataSet():          #创建数据集
    dataSet = [[0,0,0,0, 'no'],
                [0,0,0,1, 'no'],
                [0,1,0,1, 'yes'],
                [0,1,1,0, 'yes'],
                [0,0,0,0, 'no'],
                [1,0,0,0, 'no'],
                [1,0,0,1, 'no'],
                [1,1,1,1, 'yes'],
                [1,0,1,2, 'yes'],
                [2,0,1,2, 'yes'],
                [2,0,1,1, 'yes'],
                [2,1,0,1, 'yes'],
                [2,1,0,2, 'yes'],
                [2,0,0,0, 'no']]
    labels = ['年龄', '有工作否', '有房子否', '信贷情况']   #特征标签
    #change to discrete values
    return dataSet, labels

##### 计算信息熵 #####
def calcShannonEnt(dataSet):
    numEntries = len(dataSet)    # 样本数
    labelCounts = {}    # 创建一个数据字典: key 是最后一列的数值 (即标签, 也就是目标分
    #类的类别), value 是属于该类别的样本个数
    for featVec in dataSet: # 遍历整个数据集, 每次取一行
        currentLabel = featVec[-1]  #取该行最后一列的值
        if currentLabel not in labelCounts.keys(): labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1
    shannonEnt = 0.0  # 初始化信息熵
    for key in labelCounts:

```

```

prob = float(labelCounts[key])/numEntries
shannonEnt -= prob * log(prob,2) #log base 2 计算信息熵
return shannonEnt

##### 选取当前数据集下，用于划分数据集的最优特征
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1           #获取当前数据集的特征个数，最后一列是分
类标签
    baseEntropy = calcShannonEnt(dataSet)      #计算当前数据集的信息熵
    bestInfoGain = 0.0; bestFeature = -1       #初始化最优信息增益和最优的特征
    for i in range(numFeatures):               #遍历每个特征 iterate over all the features
        featList = [example[i] for example in dataSet]#获取数据集中当前特征下的所有
值
        uniqueVals = set(featList)                # 获取当前特征值，例如年龄下有 0、1、2
        newEntropy = 0.0
        for value in uniqueVals: #计算每种划分方式的信息熵
            subDataSet = splitDataSet(dataSet, i, value)
            prob = len(subDataSet)/float(len(dataSet))
            newEntropy += prob * calcShannonEnt(subDataSet)
            infoGain = baseEntropy - newEntropy      #计算信息增益
            if (infoGain > bestInfoGain):           #比较每个特征的信息增益，只要最好的信息增
益
                bestInfoGain = infoGain             #if better than current best, set to best
                bestFeature = i
    return bestFeature                         #returns an integer

##### 按给定的特征划分数据 #####
def splitDataSet(dataSet, axis, value): #axis 是 dataSet 数据集下要进行特征划分的列号例如
outlook 是 0 列，value 是该列下某个特征值，0 列中的 sunny
    retDataSet = []
    for featVec in dataSet: #遍历数据集，并抽取按 axis 的当前 value 特征进划分的数据集(不
包括 axis 列的值)
        if featVec[axis] == value: #
            reducedFeatVec = featVec[:axis]      #chop out axis used for splitting
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
            # print axis,value,reducedFeatVec
    # print retDataSet
    return retDataSet

#####该函数使用分类名称的列表，然后创建键值为 classList 中唯一值的数据字典
#####对象的存储了 classList 中每个类标签出现的频率。最后利用 operator 操作键值排序字典，
```

```

#####并返回出现次数最多的分类名称
def majorityCnt(classList):
    classCount={}
    for vote in classList:
        if vote not in classCount.keys(): classCount[vote] = 0
        classCount[vote] += 1
    sortedClassCount = sorted(classCount.iteritems(), key=operator.itemgetter(1),
reverse=True)
    return sortedClassCount[0][0]

##### 生成决策树主方法
def createTree(dataSet,labels,featLabels):
    classList = [example[-1] for example in dataSet] # 返回当前数据集下标签列所有值
    if classList.count(classList[0]) == len(classList):
        return classList[0]#当类别完全相同时则停止继续划分，直接返回该类的标签
    if len(dataSet[0]) == 1: ##遍历完所有的特征时，仍然不能将数据集划分成仅包含唯一类别的分组 dataSet
        return majorityCnt(classList) #由于无法简单的返回唯一的类标签，这里就返回出现次数最多的类别作为返回值
    bestFeat = chooseBestFeatureToSplit(dataSet) # 获取最好的分类特征索引
    bestFeatLabel = labels[bestFeat] #获取该特征的名字
    featLabels.append((bestFeatLabel))
    # 这里直接使用字典变量来存储树信息，这对于绘制树形图很重要。
    myTree = {bestFeatLabel:{}}
    del(labels[bestFeat]) #删除已经在选取的特征
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat,
value),labels,featLabels)
    return myTree

def classify(inputTree,featLabels,testVec):      #通过已有决策树和使用到的特征来对新数据
进行分类，返回类标签
    firstStr=next(iter(inputTree))
    secondDict = inputTree[firstStr]
    featIndex = featLabels.index(firstStr)+1
    for key in secondDict.keys():
        if testVec[featIndex]==key:
            if type(secondDict[key]).__name__=='dict':
                className=classify(secondDict[key],featLabels,testVec)
            else:
                className=secondDict[key]

```

```

return classLabel

if __name__ == '__main__':
    dataSet,labels=createDataSet()
    featLabels=[]
    myTree=createTree(dataSet,labels,featLabels)      #生成决策树
    #测试决策树的分类效果
    dataSet = [[0, 0, 0, 0, 'no'],
               [0, 0, 0, 1, 'no'],
               [0, 1, 0, 1, 'yes'],
               [0, 1, 1, 0, 'yes'],
               [0, 0, 0, 0, 'no'],
               [1, 0, 0, 0, 'no'],
               [1, 0, 0, 1, 'no'],
               [1, 1, 1, 1, 'yes'],
               [1, 0, 1, 2, 'yes'],
               [2, 0, 1, 2, 'yes'],
               [2, 0, 1, 1, 'yes'],
               [2, 1, 0, 1, 'yes'],
               [2, 1, 0, 2, 'yes'],
               [2, 0, 0, 0, 'no']]

    i=0
    rightnum=0
    for data in dataSet:
        i+=1
        testVec=data[0:4]
        result=classify(myTree,featLabels,testVec)
        print('数据及标签: ')
        print(data)
        print('预测标签: '+result+'\n')           #原始数据、预测数据对比
        if(result==data[-1]):
            rightnum+=1
        print('决策树: ')
        print(myTree)                           #显示决策树
        print('\n')
        print('rightratio: '+str(rightnum/len(dataSet)*100)+'%') #显示分类正确率

    print(classify(myTree,featLabels,[1,1,0,0]))

```

报告评分:

指导教师签字:

一、实验室名称：主楼 A2-412

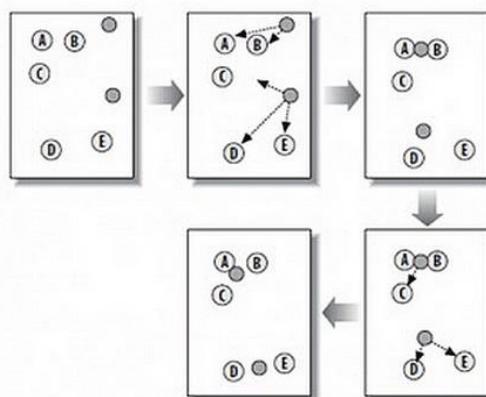
二、实验项目名称：聚类

三、实验学时：3

四、实验原理

4.1 K-Means 算法原理

- (1) 随机在图中取 K (如图 K=2) 个聚类中心。
- (2) 然后对图中的所有点求到这 K 个种子点的距离，假如点 P_i 离聚类中心 S_i 最近，那么 P_i 属于 S_i 点群。接下来，我们要移动聚类中心到属于他的“点群”的中心。
- (3) 然后重复第 2) 和第 3) 步，直到聚类中心几乎不发生变化。



4.1 K-Means 算法原理

4.2 DBSCAN 算法原理

通过检查数据集中每个对象的 ϵ -邻域来寻找聚类。如果一个点 p 的 ϵ -邻域包含多于 $MinPts$ 个对象，则创建一个 p 作为核心对象的新簇。然后，DBSCAN 反复地寻找从这些核心对象直接密度可达的对象，这个过程可能涉及一些密度可达簇的合并。当没有新的点可以被添加到任何簇时，该过程结束。具体如下：

DBSCAN 算法描述

算法 5-5 DBSCAN

输入：包含 n 个对象的数据库，半径 ϵ ，最少数目 $MinPts$ 。

输出：所有生成的簇，达到密度要求。

1. REPEAT

2. 从数据库中抽取一个未处理过的点；

3. IF 抽出的点是核心点 THEN 找出所有从该点密度可达的对象，形成一个簇

4. ELSE 抽出的点是边缘点(非核心对象)，跳出本次循环，寻找下一点；

5. UNTIL 所有点都被处理；

五、实验目的

- 1、了解聚类的基本概念、原理和一般方法
- 2、掌握聚类的基本算法
- 3、学会调用 WEKA 包处理 kmeans 聚类问题；
- 4、自己编程实现 K-Means、DBSCAN 算法；

六、实验内容

- 1、学会调用 WEKA 包处理 kmeans 聚类问题；
- 2、自己编程实现 K-Means 算法；
- 3、（选做）实现 DBSCAN 算法

七、实验器材（设备、元器件）

台式机、笔记本（MagicBook）

八、实验步骤

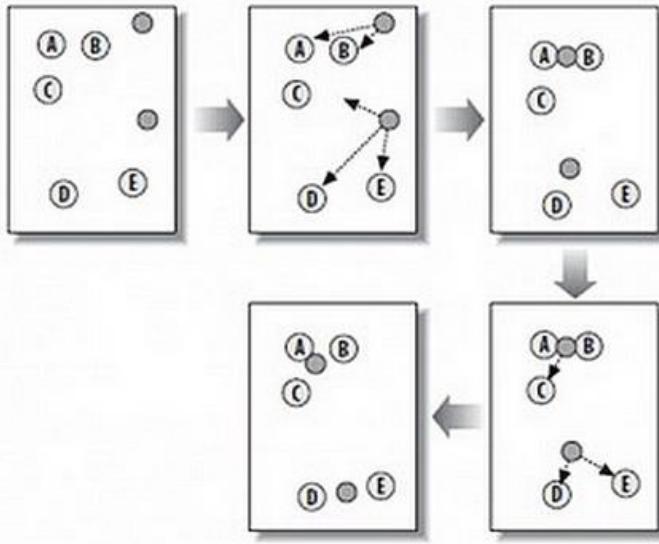
1 使用 WEKA 图形界面实现 Kmeans

2 自己动手实现 Kmeans 算法

2.1 Kmeans 算法原理

K-Means 的算法如下：

- (1) 随机在图中取 K (如图 K=2) 个聚类中心。
- (2) 然后对图中的所有点求到这 K 个种子点的距离，假如点 P_i 离聚类中心 S_i 最近，那么 P_i 属于 S_i 点群。接下来，我们要移动聚类中心到属于他的“点群”的中心。
- (3) 然后重复第 2) 和第 3) 步，直到聚类中心几乎不发生变化。



2.2 源码参考:

- (1) 加载数据集，返回一个数据矩阵
- (2) 主函数：导入数据，调用 kmeans 函数处理数据，将聚类结果可视化
- (3) Kmeans 算法实现细节

class KMeansCluster():

构造函数

计算距离时用的是欧式距离：

随机选择 k 个中心点：

聚类，更新聚类中心点直到收敛：

运行得到实验结果图

3 使用 WEKA 图形界面实现 DBSCAN

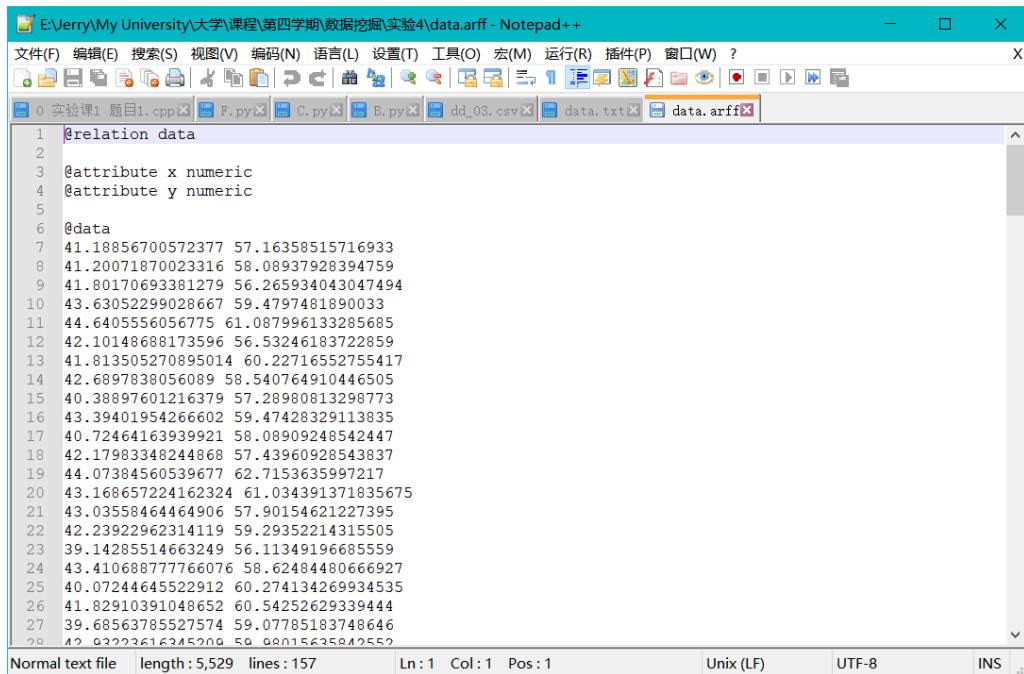
4 自己实现 DBSCAN，参考源码已给出。

- (1) 从文件中读取数据
- (2) 计算两个向量的欧式距离
- (3) 判断是否在 eps 范围内
- (4) 输入数据集，查询点的 id，半径大小，输出在 eps 范围内的点的 id
- (5) 输入数据集，分类结果，待分类点 id，簇 id，半径大小，最小点个数，输出：能否成功分类
- (6) DBScan 输入数据集，半径大小，最小点个数，输出：分类簇 id
- (7) 实现可视化
- (8) 主方法及主函数

九、实验数据及结果分析

9.1 使用 WEKA 图形界面实现 Kmeans

打开 Weka， 打开文件 data.arff。 data.arff 数据如图所示：



The screenshot shows a Notepad++ window with the file 'data.arff' open. The content of the file is as follows:

```
@relation data
@attribute x numeric
@attribute y numeric
@data
41.18856700572377 57.16358515716933
41.20071870023316 58.08937928394759
41.80170693381279 56.265934043047494
43.63052299028667 59.4797481890033
44.6405556056775 61.087996133285685
42.10148688173596 56.53246183722859
41.813505270895014 60.22716552755417
42.6897838056089 58.540764910446505
40.38897601216379 57.28980813298773
43.39401954266602 59.47428329113835
40.72464163939921 58.08909248542447
42.17983348244868 57.43960928543837
44.07384560539677 62.7153635997217
43.169657224162324 61.034391371835675
43.03558464464906 57.90154621227395
42.23922962314119 59.29352214315505
39.14285514663249 56.11349196685559
43.41068877766076 58.62484480666927
40.07244645522912 60.274134269934535
41.82910391048652 60.54252629339444
39.68563785527574 59.07785183748646
42.62223616245700 50.00156358425559
```

Normal text file length : 5,529 lines : 157 Ln:1 Col:1 Pos:1 Unix (LF) UTF-8 INS

图 9.1.1 数据集一览

使用 python， 写下如下代码， 完成数据可视化：

```
import matplotlib.pyplot as plt
f=open('data.txt')
points=[]
for i in f:
    a=i.split()
    points.append([eval(a[0]),eval(a[1])])
l=len(points)
x=[points[i][0] for i in range(l)]
y=[points[i][1] for i in range(l)]
plt.scatter(x,y,marker='.')
plt.show()
```

图 9.1.2 可视化代码

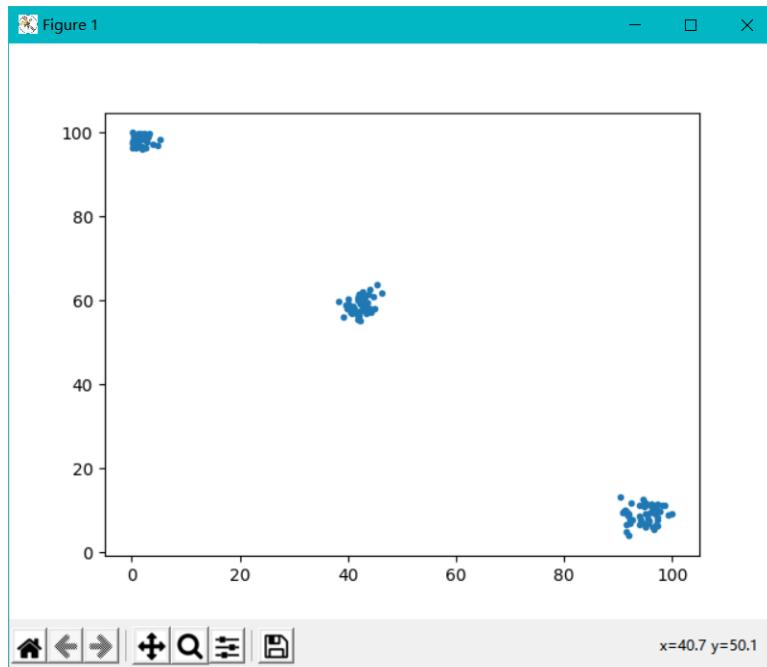


图 9.1.3 可视化结果

可见数据分布为约 3 簇。

Weka 选项卡中选择 Cluster，Clusterer 选择 SimpleKMeans，参数设置如下：

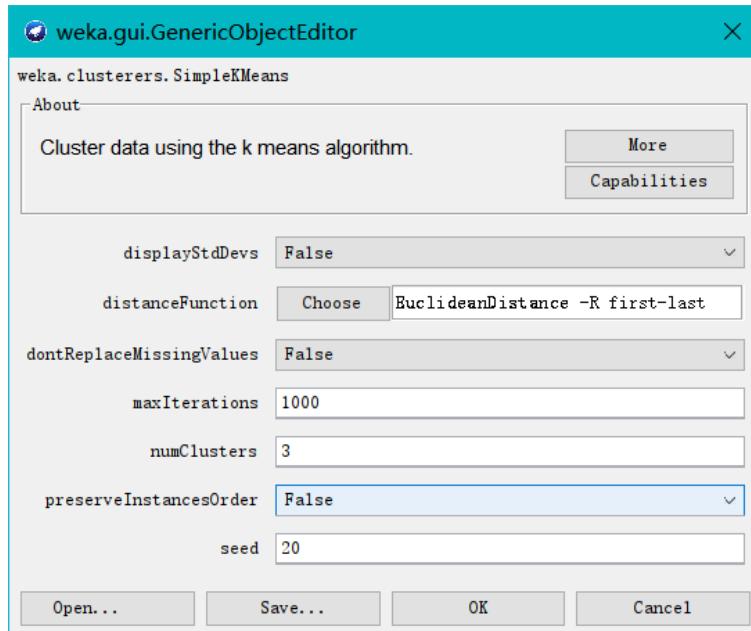


图 9.1.4 Weka 实现 Kmeans 的参数设置

(可以看到簇数 numClusters 值为 3，seed 值为 20)

事实上 seed 为随机数种子，用于决定选择哪几个点作为初始中心点，其很大程度上影响了聚类结果的正确性（Kmeans 算法非常依赖于初始点选取）。在尝试了 10、11、12 等不合适的值后，最终调整为 20 时得到了正确的聚类结果。

点击 start，运行结果如下：

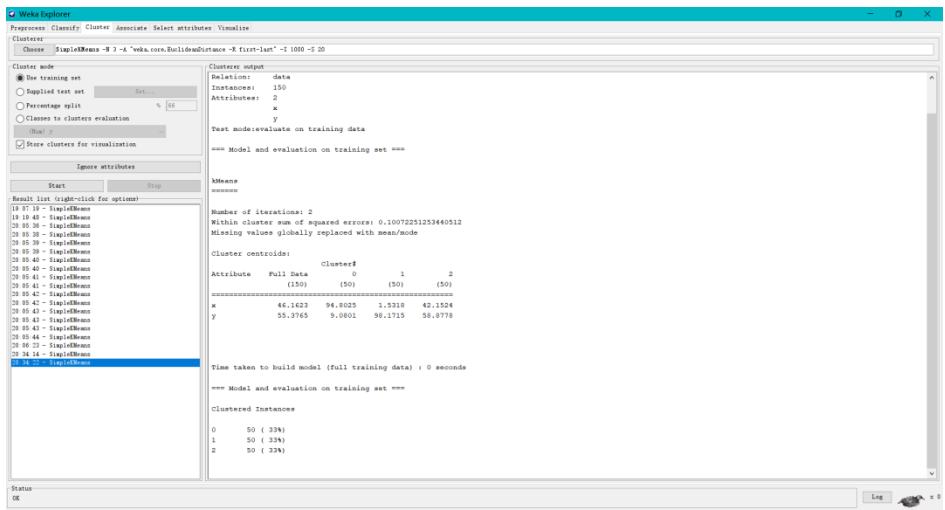


图 9.1.5 Weka 聚类运行结果

```

Cluster centroids:
          Cluster#
Attribute   Full Data      0       1       2
              (150)    (50)    (50)    (50)
=====
x           46.1623  94.8025  1.5318  42.1524
y           55.3765  9.0801   98.1715  58.8778
  
```

图 9.1.6 聚类中心点坐标

```

Clustered Instances

0      50 ( 33%)
1      50 ( 33%)
2      50 ( 33%)
  
```

图 9.1.7 聚类各簇点数

Cluster centroids:					
Attribute	Full Data	0	1	2	
	(150)	(50)	(50)	(50)	
x	46.1623	94.8025	1.5318	42.1524	
y	55.3765	9.0801	98.1715	58.8778	

Clustered Instances					
0	50 (33%)				
1	50 (33%)				
2	50 (33%)				

可见 150 个点被分成了 3 簇，每簇分别有 50、50、50 个点，每个簇的中心点分别为

[94.8025,9.0801], [1.5318,98.1715], [42.1524,58.8778]。

9.2 自己动手实现 Kmeans 算法

9.2.1 Kmeans 算法原理

K-Means 的算法如下：

- (1) 随机在图中取 K (如图 K=2) 个聚类中心。
- (2) 然后对图中的所有点求到这 K 个种子点的距离，假如点 P_i 离聚类中心 S_i 最近，那么 P_i 属于 S_i 点群。接下来，我们要移动聚类中心到属于他的“点群”的中心。
- (3) 然后重复第 2) 和第 3) 步，直到聚类中心几乎不发生变化。

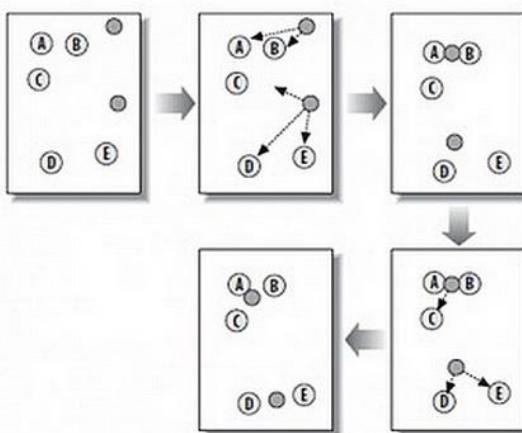


图 9.2.1 KMeans 算法原理

K-Means 主要有两个最重大的缺陷——都和初始值有关：

- (1) K 是事先给定的，这个 K 值的选定是非常难以估计的。很多时候，事先并不知道给定的数据集应该分成多少个类别才最合适。
- (2) K-Means 算法需要用初始随机种子点来搞，这个随机种子点太重要，不同的随机种子点会有得到完全不同的结果。

9.2.2 Kmeans 代码实现

结合实验原理，可以这样设计程序。

首先，核心函数为 Kmeans 函数，调用了 cluster 函数、centroid 函数，并完成中心点初始化、划分点集、更新中心点、重复至稳定等步骤，最终返回聚类结果（最终的点集划分）：

```

def Kmeans(pointlist,k):
    expointlist=pointlist
    shuffle(expointlist)
    oldcenterpoint=expointlist[:k]
    clusterpoint=cluster(oldcenterpoint,pointlist)
    newcenterpoint=centriod(clusterpoint)
    while(oldcenterpoint!=newcenterpoint):
        oldcenterpoint=newcenterpoint
        clusterpoint=cluster(oldcenterpoint,pointlist)
        newcenterpoint=centriod(clusterpoint)
    lastcluster=clusterpoint
    return lastcluster

```

图 9.2.2 Kmeans 函数

cluster 函数用于根据中心点对点集进行簇划分，其中用到了 distancebtp 函数，用于计算两点间的距离：

```

def cluster(center,pointlist):
    distancelist=[] #根据中心点对点集进行簇划分
    for i in range(len(pointlist)): #距离列表，记录每个点到不同中心的距离
        distancei=[]
        for j in range(len(center)):
            distancei.append(distancebtp(pointlist[i],center[j]))
        distancelist.append(distancei)
    nearestpoint=[] #最近点列表，记录点集中每个点的最近中心点序号
    for i in range(len(pointlist)):
        leastdisid=0
        for j in range(1,len(center)):
            if(distancelist[i][j]<distancelist[i][leastdisid]):
                leastdisid=j
        nearestpoint.append(leastdisid)
    clusterpoint=[] #记录根据该次中心点时点集的簇划分
    for i in range(len(center)):
        clusterpointi=[]
        for j in range(len(pointlist)):
            if(nearestpoint[j]==i):
                clusterpointi.append(pointlist[j])
        clusterpoint.append(clusterpointi)
    #print('该次 聚类结果: ',clusterpoint)
    return clusterpoint #返回根据该次中心点时点集的簇划分

```

图 9.2.3 cluster 函数

centriod 函数用于根据输入的簇划分返回新的中心点：

```

def centriod(clusterpoint):
    length=len(clusterpoint)
    centriodlist=[]
    pointlist=[]
    for i in range(length):
        centriodi=[0,0]
        for j in range(len(clusterpoint[i])):
            centriodi[0]+=clusterpoint[i][j][0]
            centriodi[1]+=clusterpoint[i][j][1]
            pointlist.append(clusterpoint[i][j])
        centriodi[0]/=len(clusterpoint[i])      #x均值
        centriodi[1]/=len(clusterpoint[i])      #y均值
        centriodlist.append(centriodi)          #添加中心点
    ...
    for i in range(length):
        distancei=distancebtp(centriodlist[i],pointlist[0])
        indexi=0
        for j in range(1,len(pointlist)):
            if(distancebtp(centriodlist[i],pointlist[j])<distancei):
                distancei=distancebtp(centriodlist[i],pointlist[j])
                indexi=j
        newcenter.append(pointlist[indexi])
        del pointlist[indexi]   ''
    #若将注释去掉，则为k中心算法。k均值与k中心算法的差别也仅在于此
    return centriodlist

```

图 9.2.4 centriod 函数

distancebtp(distance between points)函数用于计算两个点之间的欧氏距离：

```

def distancebtp(point1,point2):           #给定两点，计算欧几里得距离
    distance2=pow(point1[0]-point2[0],2)+pow(point1[1]-point2[1],2)
    return pow(distance2,0.5)

```

图 9.2.5 distancebtp 函数

最终完成主函数，读出文件数据，使用 Kmeans 算法，并画图：

```

if __name__=='__main__':
    #主函数，读出文件数据，使用Kmeans算法，并画图

    f = open('data.txt')
    points = []
    for i in f:
        a = i.split()
        points.append([eval(a[0]), eval(a[1])])
    #文件数据读取，得到点集

    k=3
    colorlist = ['red', 'peru', 'gold', 'lawngreen', 'cyan', 'navy', 'purple', 'hotpink']  #颜色列表
    lastcluster = Kmeans(points, k)           #Kmeans算法
    #print(lastcluster)
    for i in range(len(lastcluster)):
        plt.scatter([lastcluster[i][j][0] for j in range(len(lastcluster[i]))],[lastcluster[i][j][1] for j in range(len(lastcluster[i]))],marker='*')
    plt.show()

```

图 9.2.6 主函数

(源码详见附件)

9.2.3 手动实现 Kmeans 的运行结果

运行上述代码，可以得到如下运行结果和生成图像：

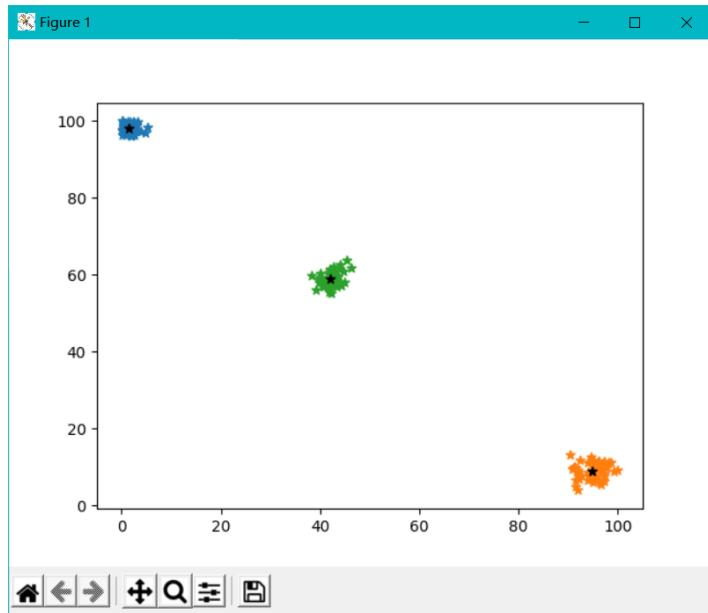


图 9.2.7 聚类结果图及各簇中心点

```
D:\Python\Python3.8.8\python.exe "E:/Jerry/My University/大学/课程/第四学期/数据挖掘/实验4/KmeanCluster.py"
第1个簇的中心点:
[1.5318467520330943, 98.17146850174129]
含有点数: 50

第2个簇的中心点:
[94.80254740755093, 9.080123939210864]
含有点数: 50

第3个簇的中心点:
[42.15235802904096, 58.87781764760191]
含有点数: 50

Process finished with exit code 0
```

图 9.2.8 各簇中心点及含有的中心点个数

可见：

第 1 个簇的中心点：

[1.5318467520330943, 98.17146850174129]

含有点数：50

第 2 个簇的中心点：

[94.80254740755093, 9.080123939210864]

含有点数：50

第 3 个簇的中心点：

[42.15235802904096, 58.87781764760191]

含有点数：50

将此结果与 9.1 使用 WEKA 的运行结果（如下图）对比可见，两者几乎完全一样（仅有保留小数位数不同的差异），故程序编写成功！

Cluster centroids:					
Attribute	Full Data	0 (150)	1 (50)	2 (50)	
x	46.1623	94.8025	1.5318	42.1524	
y	55.3765	9.0801	98.1715	58.8778	
Clustered Instances					
0	50 (33%)				
1	50 (33%)				
2	50 (33%)				

图 9.2.9 9.1WEKA 运行结果

9.2.4 额外 手动实现 k 中心算法

k 中心算法是对 Kmeans 算法的优化，减小了初始点选取对结果造成的影响，同时时间复杂度更高。需要修改的 centriod 核心代码下：

```

def centriod(clusterpoint):          #输入簇划分, 返回k个均值点
    length=len(clusterpoint)
    centriodlist=[]                  #中心点列表
    pointlist=[]
    for i in range(length):
        centriodi=[0,0]
        for j in range(len(clusterpoint[i])):
            centriodi[0]+=clusterpoint[i][j][0]
            centriodi[1]+=clusterpoint[i][j][1]
        pointlist.append(clusterpoint[i][j])
        centriodi[0]/=len(clusterpoint[i])      #x均值
        centriodi[1]/=len(clusterpoint[i])      #y均值
        centriodlist.append(centriodi)          #添加中心点
    newcenter=[]
    for i in range(length):
        distancei=distancebtw(centriodlist[i],pointlist[0])
        indexi=0
        for j in range(i,len(pointlist)):
            if(distancebtw(centriodlist[i],pointlist[j])<distancei):
                distancei=distancebtw(centriodlist[i],pointlist[j])
                indexi=j
        newcenter.append(pointlist[indexi])
        del pointlist[indexi]                 #若将注释去掉, 则为k中心算法. k均值与k中心算法的差别也仅在于此
    return newcenter                      #返回给定簇划分时的新中心点

```

图 9.2.10 centriod 函数

使得到簇的中心点后将中心点更新为距其最近的数据点，也即完成了代码改进。运行结果如下：

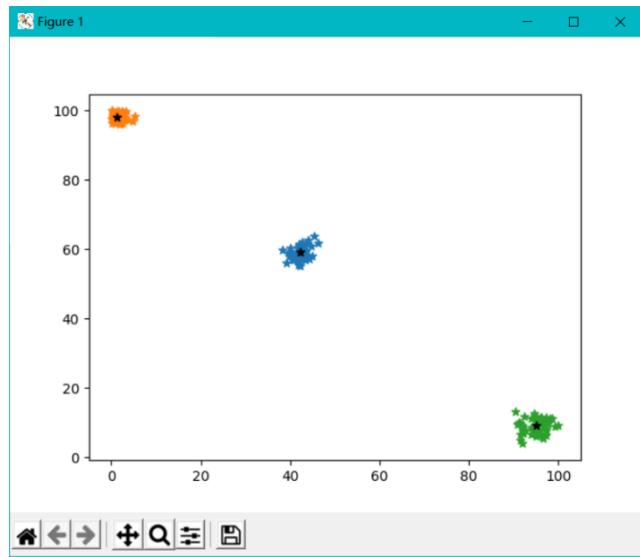


图 9.2.11 聚类结果图及各簇中心点

```
D:\Python\Python3.8.8\python.exe "E:/Jerry/My University/大学/课程/第四学期/数据挖掘/实验4/KmeanMoids.py"
第1个簇的中心点:
[42.23922962314119, 59.29352214315505]
含有点数: 50

第2个簇的中心点:
[1.2122038390520575, 98.14917158448738]
含有点数: 50

第3个簇的中心点:
[94.97971281850847, 9.204812228137186]
含有点数: 50

Process finished with exit code 0
```

图 9.2.12 各簇中心点及含有的中心点个数

第 1 个簇的中心点:

[42.23922962314119, 59.29352214315505]

含有点数: 50

第 2 个簇的中心点:

[1.2122038390520575, 98.14917158448738]

含有点数: 50

第 3 个簇的中心点:

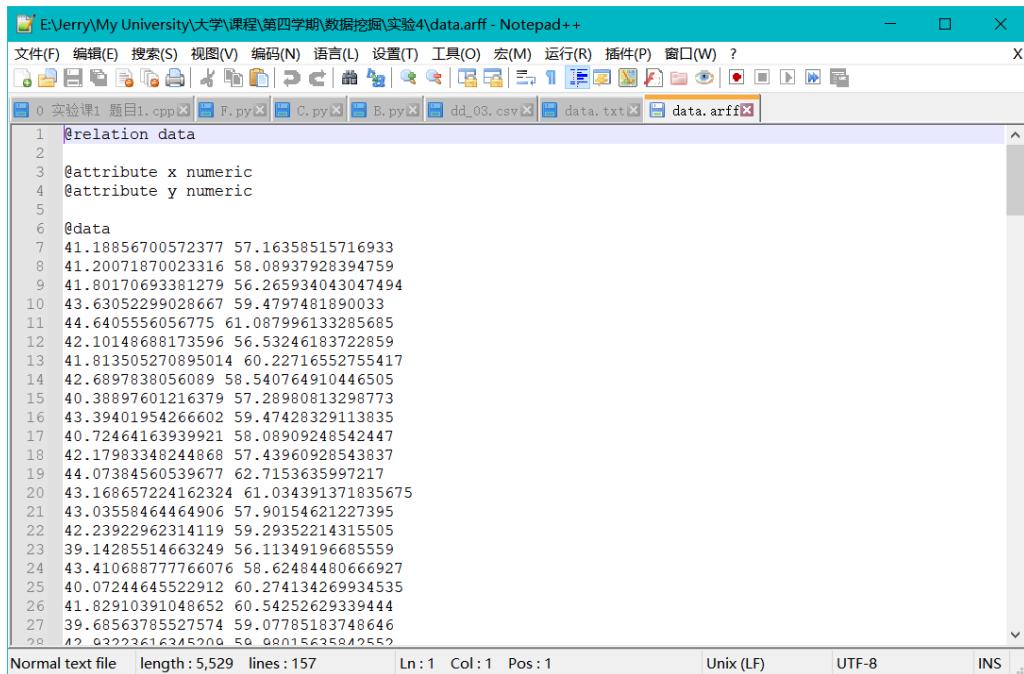
[94.97971281850847, 9.204812228137186]

含有点数: 50

将 k 中心运行结果与 Kmeans 算法运行结果对比，可见，分类结果是一致的，这也验证了代码正确性。

9.3 使用 WEKA 图形界面实现 DBSCAN

仍然使用 data.arff 数据集。



The screenshot shows a Notepad++ window with the file "data.arff" open. The content of the file is as follows:

```
@relation data
@attribute x numeric
@attribute y numeric
@data
1.18856700572377 57.16358515716933
1.20071870023316 58.08937928394759
1.80170693381279 56.265934043047494
10.63052299028667 59.4797481890033
11.6405556056775 61.087996133285685
12.10148688173596 56.53246183722859
13.813505270895014 60.22716552755417
14.6897838056089 58.540764910446505
15.38897601216379 57.28980813298773
16.39401954266602 59.47428329113835
17.72464163939921 58.08909248542447
18.17983348244868 57.43960928543837
19.07384560539677 62.7153635997217
20.169657224162324 61.034391371835675
21.03558464464906 57.90154621227395
22.23922962314119 59.29352214315505
23.39.14285514663249 56.11349196685559
24.41.01068877766076 58.62484480666927
25.40.07244645522912 60.274134269934535
26.41.82910391048652 60.54252629339444
27.39.68563785527574 59.07785183748646
28.12.02223616245700 50.00156258025552
```

Normal text file length: 5,529 lines: 157 Ln:1 Col:1 Pos:1 Unix (LF) UTF-8 INS

图 9.3.1 数据集一览

Weka 选项卡中选择 Cluster，Clusterer 选择 DBSCAN，参数设置如下：

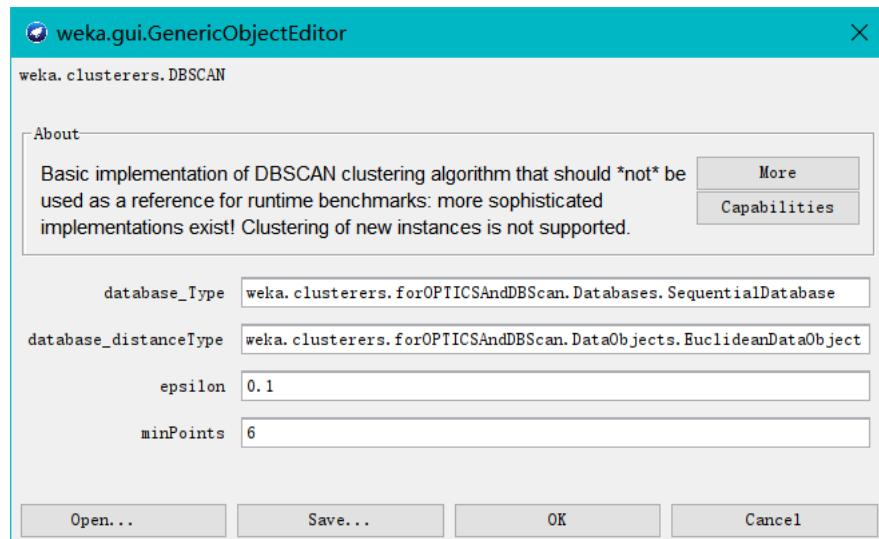


图 9.3.2 Weka 实现 DBSCAN 的参数设置

(可以看到邻域半径 epsilon 值为 0.1, minPoints 值为 6)

epsilon、minPoints 的选取影响输出结果，在调试后得到了成功的输出。

点击 start，运行结果如下：

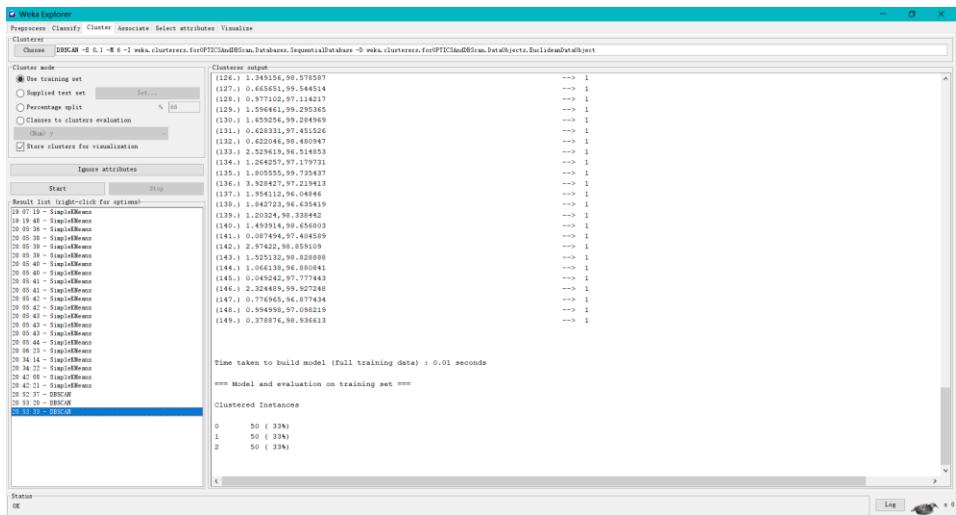


图 9.3.3 Weka 实现 DBSCAN 的运行结果

Clustered Instances

0	50 (33%)
1	50 (33%)
2	50 (33%)

图 9.3.3 DBSCAN 簇划分结果

可见 150 个点被分成 3 簇，分别有 50、50 和 50 个点。这与我们在 9.1、9.2 中得到的结果完全一致，运行成功。

9.4 自己实现 DBSCAN

9.4.1 程序设计

设计读取数据的函数 loadDataSet，如下：

```
def loadDataSet(fileName):
    f = open(fileName)
    dataSet = []
    for i in f:
        a = i.split()
        dataSet.append([eval(a[0]), eval(a[1])])
    return dataSet
```

图 9.4.1 loadDataSet 函数

返回点的列表。

距离计算函数，dist，用于计算两点间欧几里得距离。

```

def dist(a, b):
    return math.sqrt(np.power(a - b, 2).sum())

```

图 9.4.2 dist 函数

eps_neighbor 函数，用于判断 a, b 两点是否在彼此的 epsilon 范围内。此处用到了刚才的 dist 函数。

```

def eps_neighbor(a, b, eps):
    return dist(a, b) < eps

```

图 9.4.3 eps_neighbor 函数

寻找邻居函数 region_query，找到并返回点周围邻域半径里的相邻点序号。

```

def region_query(data, pointId, eps):
    nPoints = data.shape[1]
    seeds = []
    for i in range(nPoints):
        if eps_neighbor(data[:, pointId], data[:, i], eps):
            seeds.append(i)
    return seeds

```

图 9.4.4 region_query 函数

簇扩张函数 expend_cluster，根据点序号扩大其所在的簇，如下：

```

def expend_cluster(data, clusterResult, pointId, clusterId, eps, minPts):
    seeds = region_query(data, pointId, eps)
    if len(seeds) < minPts:
        clusterResult[pointId] = NOISE
        return False
    else:
        clusterResult[pointId] = clusterId
        for seedId in seeds:
            clusterResult[seedId] = clusterId
        while len(seeds) > 0:
            currentPoint = seeds[0]
            queryResults = region_query(data, currentPoint, eps)
            if len(queryResults) >= minPts:
                for i in range(len(queryResults)):
                    resultPoint = queryResults[i]
                    if clusterResult[resultPoint] == UNCLASSIFIED:
                        seeds.append(resultPoint)
                        clusterResult[resultPoint] = clusterId
                    elif clusterResult[resultPoint] == NOISE:
                        clusterResult[resultPoint] = clusterId
            seeds = seeds[1:]
        return True

```

图 9.4.5 expend_cluster 函数

然后是核心函数 DBSCAN 函数，调用了以上函数，并根据输入的点集得到簇划分和簇个数并返回。

```

def dbscan(data, eps, minPts):
    #DBSCAN核心函数，调度其余函数
    clusterId = 1
    nPoints = data.shape[1]
    clusterResult = [UNCLASSIFIED] * nPoints
    #初始化
    for pointId in range(nPoints):
        #point = data[:, pointId]
        if clusterResult[pointId] == UNCLASSIFIED:
            #对每个点完成搜索
            if expend_cluster(data, clusterResult, pointId, clusterId, eps, minPts):
                clusterId = clusterId + 1
    return clusterResult, clusterId - 1

```

图 9.4.6 dbscan 函数

实现可视化的函数 plotFeature，根据点集、簇划分、簇个数实现绘图：

```

def plotFeature(data, clusters, clusterNum):
    #可视化
    matClusters = np.mat(clusters).transpose()
    fig = plt.figure()
    scatterColors = ['black', 'blue', 'green', 'yellow', 'red', 'purple', 'orange', 'brown']
    ax = fig.add_subplot(111)
    for i in range(clusterNum + 1):
        print(i)
        colorStyle = scatterColors[i % len(scatterColors)]
        subCluster = data[:, np.nonzero(matClusters[:, 0].A == i)]
        ax.scatter(subCluster[0, :].flatten().A[0], subCluster[1, :].flatten().A[0], c=colorStyle, s=50)

```

图 9.4.7 plotFeature 函数

最后是主方法及主函数，用于完成对数据集实践 DBSCAN 算法并输出运行结果、可视化：

```

def main():
    dataSet = loadDataSet('data.txt')
    dataSet = np.mat(dataSet).transpose()
    clusters, clusterNum = dbscan(dataSet, 8, 1)
    cnum=len(set(clusters))
    for i in range(cnum):
        l=[]
        for j in range(len(clusters)):
            if clusters[j]==i+1:
                l.append(j)
        x=[dataSet[0,j] for j in l]
        xm=sum(x)/len(x)
        y = [dataSet[1, j] for j in l]
        ym = sum(y) / len(y)
        print('第' + str(i + 1) + '个簇的中心点: ')
        print([xm,ym])
        print('含有点数: ' + str(len(l)))
        print('')
    print("cluster Numbers=", clusterNum)
    plotFeature(dataSet, clusters, clusterNum)

```

图 9.4.8 main 函数

```

if __name__ == '__main__':
    start = time.time()
    main()
    end = time.time()
    print('finish all in %s' % str(end - start))      #显示用时
    plt.show()

```

图 9.4.9 程序入口

9.4.2 运行结果

运行以上程序，能够得到如下的运行结果：

```
D:\Python\Python3.8.8\python.exe "E:/Jerry/My University/大学/课程/第四学期/数据挖掘/实验4/DBSCAN.py"
第1个簇的中心点:
[42.15235802904096, 58.877817647601915]
含有点数: 50

第2个簇的中心点:
[94.80254740755096, 9.080123939210866]
含有点数: 50

第3个簇的中心点:
[1.5318467520330943, 98.17146850174133]
含有点数: 50

cluster Numbers= 3
finish all in 0.6535656452178955

Process finished with exit code 0
```

图 9.4.10 程序运行结果 1

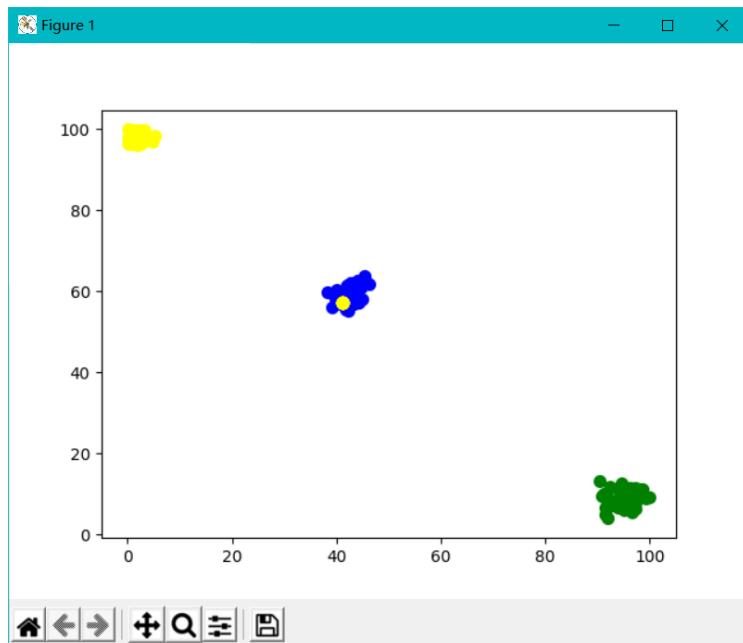


图 9.4.10 程序运行结果 2

不管是输出还是可视化，都能看到分成了 3 个簇，效果优良。其中：

第 1 个簇的中心点：

[42.15235802904096, 58.877817647601915]

含有点数：50

第 2 个簇的中心点：

[94.80254740755096, 9.080123939210866]

含有点数: 50

第 3 个簇的中心点:

[1.5318467520330943, 98.17146850174133]

含有点数: 50

这与我们使用 Kmeans、k 中心算法以及 WEKA 图形界面得到的结果都是一致的。实验成功!

程序运行时间共 0.637052059173584 秒。

十、实验结论

成功利用 Weka 完成了 Kmeans 算法、DBSCAN 算法,用 Python 自己动手实现了 Kmeans 算法、DBSCAN 算法, 并修改代码尝试了 K 中心算法。

Kmeans 算法十分依赖于初始点选取, WEKA 和 python 代码编写过程中都显示了这一问题, 相比之下 K 中心算法和 DBSCAN 针对 data 数据集有更好的聚类效果。

十一、总结及心得体会

成功利用 Weka 完成了 Kmeans 算法、DBSCAN 算法,用 Python 自己动手实现了 Kmeans 算法、DBSCAN 算法, 并修改代码尝试了 K 中心算法。

使用 Weka 实现和自己动手实践得到了同样的聚类结果, 符合实验预期。通过这次实验我对数据的认识更加深刻, 提高了动手能力, 也更加理解了 Kmeans 算法、K 中心算法、DBSCAN 算法, 并对数据挖掘有了更浓厚的兴趣, 希望以后能多进行类似实验。

十二、对本实验过程及方法、手段的改进建议

建议提前发出 PPT 及指导书, 以便学生预习。

提供多种多样的数据集（或来源）, 以便测试。

十三、源码

数据集可视化代码：

```
import matplotlib.pyplot as plt
f=open('data.txt')
points=[]
for i in f:
    a=i.split()
    points.append([eval(a[0]),eval(a[1])])
l=len(points)
print(l)
x=[points[i][0] for i in range(l)]
y=[points[i][1] for i in range(l)]
plt.scatter(x,y,marker='.')
plt.show()
```

自己动手实现 Kmeans 算法：

```
from random import*
import matplotlib.pyplot as plt

def Kmeans(pointlist,k):                                #K 均值聚类的核心函数
    expointlist=pointlist
    shuffle(expointlist)
    oldcenterpoint=expointlist[:k]                      #选取 k 个点作为初始中心点
    clusterpoint=cluster(oldcenterpoint,pointlist)      #根据旧中心点对点集进行簇划分
    newcenterpoint=centriod(clusterpoint)                #找到划分好的簇的新中心点
    while(oldcenterpoint!=newcenterpoint):               #当新旧中心点不一样时继续更新
        oldcenterpoint=newcenterpoint                   #旧中心点
        clusterpoint=cluster(oldcenterpoint,pointlist)   #根据旧中心点对点集进行簇划分
        newcenterpoint=centriod(clusterpoint)              #新中心点
    lastcluster=clusterpoint
    return lastcluster                                    #返回聚类结果

def cluster(center,pointlist):                          #根据中心点对点集进行簇划分
    distancelist=[]                                     #距离列表，记录每个点到不同中心的距离
    for i in range(len(pointlist)):
        distancei=[]
        for j in range(len(center)):
            distancei.append(distancebtw(pointlist[i],center[j]))
        distancelist.append(distancei)
    nearestpoint=[]                                     #最近点列表，记录点集中
```

```

每个点的最近中心点序号
for i in range(len(pointlist)):
    leastdisid=0
    for j in range(1,len(center)):
        if(distanancelist[i][j]<distanancelist[i][leastdisid]):
            leastdisid=j
    nearestpoint.append(leastdisid)
clusterpoint=[]                                #记录根据该次中心点时点
集的簇划分
for i in range(len(center)):
    clusterpointi=[]
    for j in range(len(pointlist)):
        if(nearestpoint[j]==i):
            clusterpointi.append(pointlist[j])
    clusterpoint.append(clusterpointi)
#print('该次 聚类结果: ',clusterpoint)
return clusterpoint                            #返回根据该次中心点时点
集的簇划分

def centriod(clusterpoint):                    #输入簇划分, 返回 k 个均值点
    length=len(clusterpoint)
    centriodlist=[]                           #中心点列表
    pointlist=[]
    for i in range(length):
        centriodi=[0,0]
        for j in range(len(clusterpoint[i])):
            centriodi[0]+=clusterpoint[i][j][0]
            centriodi[1]+=clusterpoint[i][j][1]
            pointlist.append(clusterpoint[i][j])
        centriodi[0]/=len(clusterpoint[i])      #x 均值
        centriodi[1]/=len(clusterpoint[i])      #y 均值
        centriodlist.append(centriodi)          #添加中心点
    """
    for i in range(length):
        distancei=distancebtp(centriodlist[i],pointlist[0])
        indexi=0
        for j in range(1,len(pointlist)):
            if(distancebtp(centriodlist[i],pointlist[j])<distancei):
                distancei=distancebtp(centriodlist[i],pointlist[j])
                indexi=j
        newcenter.append(pointlist[indexi])
        del pointlist[indexi]                  "#若将注释去掉, 则为 k 中心算法。
    k 均值与 k 中心算法的差别也仅在于此

```

```

    return centriodlist #返回给定簇划分时的新中心点

def distancebtp(point1,point2): #给定两点，计算欧几里得距离
    distance2=pow(point1[0]-point2[0],2)+pow(point1[1]-point2[1],2)
    return pow(distance2,0.5)

if(__name__=='__main__'): #主函数，读出文件数据，使用
    Kmeans 算法，并画图

    f = open('data.txt')
    points = []
    for i in f:
        a = i.split()
        points.append([eval(a[0]), eval(a[1])])
    #文件数据读取，得到点集

    k=3
    colorlist = ['red', 'peru', 'gold', 'lawngreen', 'cyan', 'navy', 'purple', 'hotpink'] #颜色列表
    lastcluster = Kmeans(points, k)
    #Kmeans 算法
    center=centriod(lastcluster)
    #聚类中心点
    #print(lastcluster)
    for i in range(len(lastcluster)):
        #不同簇使用不同颜色，绘图
        plt.scatter([lastcluster[i][j][0] for j in
range(len(lastcluster[i]))],[lastcluster[i][j][1] for j in
range(len(lastcluster[i]))],marker='*')
        plt.scatter([center[j][0] for j in range(len(center))],[center[j][1] for j in
range(len(center))],marker='*',color='k') #中心点绘制
    plt.show()
    #展示图形
    for i in range(k):
        #展示聚类结果
        print('第'+str(i+1)+'个簇的中心点: ')
        print(center[i])
        print('含有点数: '+str(len(lastcluster[i])))
        print('')

```

自己动手实现 KMedoids 算法：

```

from random import*
import matplotlib.pyplot as plt

```

```

def Kmeans(pointlist,k):                                #K 均值聚类的核心函数
    expointlist=pointlist
    shuffle(expointlist)
    oldcenterpoint=expointlist[:k]                      #选取 k 个点作为初始中心点
    clusterpoint=cluster(oldcenterpoint,pointlist)      #根据旧中心点对点集进行簇划分
    newcenterpoint=centriod(clusterpoint)                #找到划分好的簇的新中心点
    while(oldcenterpoint!=newcenterpoint):               #当新旧中心点不一样时继续更新
        oldcenterpoint=newcenterpoint                   #旧中心点
        clusterpoint=cluster(oldcenterpoint,pointlist)   #根据旧中心点对点集进行簇划分
        newcenterpoint=centriod(clusterpoint)             #新中心点
    lastcluster=clusterpoint
    return lastcluster                                    #返回聚类结果

def cluster(center,pointlist):                         #根据中心点对点集进行簇划分
    distancelist=[]                                     #距离列表，记录每个点到不同中心的距离
    for i in range(len(pointlist)):
        distancei=[]
        for j in range(len(center)):
            distancei.append(distancebtw(pointlist[i],center[j]))
        distancelist.append(distancei)
    nearestpoint=[]                                     #最近点列表，记录点集中每个点的最近中心点序号
    for i in range(len(pointlist)):
        leastdisid=0
        for j in range(1,len(center)):
            if(distancelist[i][j]<distancelist[i][leastdisid]):
                leastdisid=j
        nearestpoint.append(leastdisid)
    clusterpoint=[]                                     #记录根据该次中心点时点集的簇划分
    for i in range(len(center)):
        clusterpointi=[]
        for j in range(len(pointlist)):
            if(nearestpoint[j]==i):
                clusterpointi.append(pointlist[j])
        clusterpoint.append(clusterpointi)
    #print('该次 聚类结果: ',clusterpoint)
    return clusterpoint                                #返回根据该次中心点时点集的簇划分

def centriod(clusterpoint):                          #输入簇划分，返回 k 个均值点

```

```

length=len(clusterpoint)
centriodlist=[]
#中心点列表
pointlist=[]
for i in range(length):
    centriodi=[0,0]
    for j in range(len(clusterpoint[i])):
        centriodi[0]+=clusterpoint[i][j][0]
        centriodi[1]+=clusterpoint[i][j][1]
        pointlist.append(clusterpoint[i][j])
    centriodi[0]/=len(clusterpoint[i])           #x 均值
    centriodi[1]/=len(clusterpoint[i])           #y 均值
    centriodlist.append(centriodi)               #添加中心点
newcenter=[]
for i in range(length):
    distancei=distancebtp(centriodlist[i],pointlist[0])
    indexi=0
    for j in range(1,len(pointlist)):
        if(distancebtp(centriodlist[i],pointlist[j])<distancei):
            distancei=distancebtp(centriodlist[i],pointlist[j])
            indexi=j
    newcenter.append(pointlist[indexi])
    del pointlist[indexi]                      #若将注释去掉，则为 k 中心算法。
k 均值与 k 中心算法的差别也仅在于此
return newcenter                                #返回给定簇划分时的新中心
点

```

```

def distancebtp(point1,point2):                  #给定两点，计算欧几里得距离
    distance2=pow(point1[0]-point2[0],2)+pow(point1[1]-point2[1],2)
    return pow(distance2,0.5)

```

```

if(__name__=='__main__'):                         #主函数，读出文件数据，使用
Kmeans 算法，并画图

```

```

f = open('data.txt')
points = []
for i in f:
    a = i.split()
    points.append([eval(a[0]), eval(a[1])])
#文件数据读取，得到点集

k=3
colorlist = ['red', 'peru', 'gold', 'lawngreen', 'cyan', 'navy', 'purple', 'hotpink']      #
颜色列表

```

```

lastcluster = Kmeans(points, k)
#Kmeans 算法
    center=centriod(lastcluster)
#聚类中心点
    #print(lastcluster)
    for i in range(len(lastcluster)):
#不同簇使用不同颜色，绘图
        plt.scatter([lastcluster[i][j][0] for j in
range(len(lastcluster[i]))],[lastcluster[i][j][1] for j in
range(len(lastcluster[i]))],marker='*')
        plt.scatter([center[j][0] for j in range(len(center))],[center[j][1] for j in
range(len(center))],marker='*',color='k')      #中心点绘制
    plt.show()
#展示图形
    for i in range(k):
#展示聚类结果
        print('第'+str(i+1)+'个簇的中心点: ')
        print(center[i])
        print('含有点数: '+str(len(lastcluster[i])))
        print('')

```

自己动手实现 DBSCAN 算法：

```

import numpy as np
import matplotlib.pyplot as plt
import math
import time

UNCLASSIFIED = False
NOISE = 0

def loadDataSet(fileName):
    f = open(fileName)                      #读取数据
    dataSet = []                           #打开文件夹
    for i in f:                            #数据集初始化
        a = i.split()                      #将一行的两个数据分隔
开
        dataSet.append([eval(a[0]), eval(a[1])]) #字符串转数字并存储点坐标
    return dataSet

def dist(a, b):                          #计算欧几里得距离
    return math.sqrt(np.power(a - b, 2).sum())

```

```

def eps_neighbor(a, b, eps): #判断 a 和 b 的距离是否满足
    eps 阈值
        return dist(a, b) < eps

def region_query(data, pointId, eps): #找到点周围邻域半径里的相邻
    点序号
        nPoints = data.shape[1] #返回数据点个数
        seeds = []
        for i in range(nPoints):
            #print(i,data[:, i])
            if eps_neighbor(data[:, pointId], data[:, i], eps):
                seeds.append(i)
        return seeds

def expend_cluster(data, clusterResult, pointId, clusterId, eps, minPts): #根据点序号扩
    大其所在的簇
        seeds = region_query(data, pointId, eps) #返回 pointId 对应点周围所有在
        eps 邻域内的点序号
        if len(seeds) < minPts: #考察该点密度大小,
            clusterResult[pointId] = NOISE
            return False
        若太小则返回错误
        else:
            clusterResult[pointId] = clusterId #否则， 标记为该点的簇号
            for seedId in seeds:
                clusterResult[seedId] = clusterId #且邻域内所有点都为同样簇
            while len(seeds) > 0:
                currentPoint = seeds[0]
                queryResults = region_query(data, currentPoint, eps) #返回邻域内所
            有点的临近点个数
                if len(queryResults) >= minPts: #考察邻域内所
                    有点是否满足密度条件
                        for i in range(len(queryResults)):
                            resultPoint = queryResults[i] #点序号
                            if clusterResult[resultPoint] == UNCLASSIFIED: #未分类则划分类别
                                seeds.append(resultPoint)
                                clusterResult[resultPoint] = clusterId
                            elif clusterResult[resultPoint] == NOISE: #噪声点
                                clusterResult[resultPoint] = clusterId
                seeds = seeds[1:]
            return True

```

```

def dbscan(data, eps, minPts):
    #DBSCAN 核心函数，调度其余函数
    clusterId = 1
    nPoints = data.shape[1]
    clusterResult = [UNCLASSIFIED] * nPoints
    #初始化
    for pointId in range(nPoints):
        #对每个点完
        成搜索
        #point = data[:, pointId]
        if clusterResult[pointId] == UNCLASSIFIED:
            #如果该点未
            分类，则进入下一步判断
            if expend_cluster(data, clusterResult, pointId, clusterId, eps, minPts):
                clusterId = clusterId + 1
    return clusterResult, clusterId - 1

def plotFeature(data, clusters, clusterNum):
    #可视化
    matClusters = np.mat(clusters).transpose()
    fig = plt.figure()
    scatterColors = ['black', 'blue', 'green', 'yellow', 'red', 'purple', 'orange', 'brown']
    ax = fig.add_subplot(111)
    for i in range(clusterNum + 1):
        colorSyle = scatterColors[i % len(scatterColors)]
        subCluster = data[:, np.nonzero(matClusters[:, 0].A == i)]
        ax.scatter(subCluster[0, :].flatten().A[0], subCluster[1, :].flatten().A[0],
        c=colorSyle, s=50)

def main():
    dataSet = loadDataSet('data.txt')
    #读取数据
    dataSet = np.mat(dataSet).transpose()
    #生成矩阵并转置，得到所有横
    坐标与纵坐标
    clusters, clusterNum = dbscan(dataSet, 8, 1)
    #得到分好的簇及聚类簇数
    cnum=len(set(clusters))
    for i in range(cnum):
        l=[]
        for j in range(len(clusters)):
            if clusters[j]==i+1:
                l.append(j)
        x=[dataSet[0,j] for j in l]
        xm=sum(x)/len(x)
        y = [dataSet[1, j] for j in l]
        ym = sum(y) / len(y)
        print('第' + str(i + 1) + '个簇的中心点: ')
        print([xm,ym])
        print('含有点数: ' + str(len(l)))

```

```
print()
print("cluster Numbers=", clusterNum)          #显示信息
plotFeature(dataSet, clusters, clusterNum)       #绘图

if __name__ == '__main__':
    start = time.time()
    main()
    end = time.time()
    print('finish all in %s' % str(end - start))   #显示用时
    plt.show()
```

报告评分:

指导教师签字: