合肥工业大学

《数据挖掘》实验报告

学　　号 2017218007

姓　　名 文　华

专 业 班 级 物联网工程17-2班

指 导 老 师 吕俊伟、张玉红

合肥工业大学

2020年10月

目 录

1 实验 1：基于 UCI soybean Dataset 的分类任务

1.1 实验目的

- 熟练掌握基本的数据预处理技术；
- 学会运用决策树、随机森林等方法解决分类任务。

1.2 实验任务

基于 Molecular Biology DataSet 完成分类任务，kNN、决策树、多层感知器、朴素贝叶斯、SVM、随机森林、bagging 方法任选或组合，且不限于上述方法和策略，允许有预处理步骤。

1.3 实验环境

- **硬件：**Dell G3 3579 笔记本
- 软件：

  OS：Windows 10 Pro N for Workstations

  平台工具：PyCharm 2019.3.4 (Professional Edition)、Python 3.7.4、OriginPro 2018(64-bit)

1.4 实验内容

1.4.1 数据清洗

数据集中有的数据项严重缺失，为了方便下一步工作，需要剔除训练数据中缺失值大于 90%的列。具体操作是构建函数 drop_col，以待剔除数据集、列名与阈值为参数，在导入训练数据时调用其进行筛除。实现代码如下所示：

```
def drop_col(df, col_name, cutoff=0.9):
    n = len(df)
    cnt = df[col_name].count()
    if (float(cnt) / n) < cutoff:
        df.drop(col_name, axis=1, inplace=True)
```

1.4.2 数据导入

原始数据集中的数据为 csv 格式，使用 Python 的第三方库 pandas 的 csv 读取方法可以方便地处理。如 1.4.1 所述，严重缺失数据的列（仅对于训练数据）应当删除，数据缺失不严重的使用 sklearn 的 SimpleImputer 进行补全，本次实验 SimpleImputer 的参数设置为 most_frequent。最后将格式化的数据存入 df 中，并返回 df 的值。实现代码如下所示：

```
url = "data/soybean-large.data"
dataset = pd.read_csv(url, names=names)
dataset = dataset.replace({'?': np.nan})
for item in dataset.columns.values:
    drop_col(dataset, item, cutoff=0.8)
df1 = dataset.iloc[:, 1:]
df2 = dataset.iloc[:, :1]
imr = SimpleImputer(strategy='most_frequent')
imr = imr.fit(df1)
imputed_data = imr.transform(df1.values)
df = pd.DataFrame(imputed_data)
df = pd.concat([df2, df], axis=1)
return df
```

1.4.3 kNN 算法

实验过程中主要使用 sklearn 函数包来实现 kNN 方法，同时为了更好地调节 kNN 算法的参数，本次实验实现了一个 choose_best_k_to_knn 函数用于选取最大值，即 best_k 以及对应的下标。首先，从 sklearn.neighbors 函数包中导入 KNeighborsClassifier 函数。然后，使用 best 作为参数初始化 KNeighborsClassifier。最终，将分割好的数据分别作为第一个与第二个参数，对 kNN 模型进行训练。实现代码如下所示：

```
best_k, max_value = choose_best_k_to_knn(x_train, y_train, x_validation, y_validation)
knn = KNeighborsClassifier(n_neighbors=best_k)
knn.fit(x_train, y_train)
```

值得一提的是，本次实验所用的 scikit-learn（简称 sklearn）是目前最受欢迎，也是功能最强大的一个用于机器学习的 Python 库件。它广泛地支持各种分类、聚类以及回归分析方法比如支持向量机、随机森林、DBSCAN 等等，由于其强大的功能、优异的拓展性以及易用性，目前受到了很多数据科学从业者的欢迎，也是业界相当著名的一个开源项目之一。kNN 的实验结果如图 1.4.1 所示。

```
KNN accuracy:       0.8225806451612904
KNN precision:      0.9215686274509803
KNN recall:         0.8718487394957983
KNN f1 score:       0.8686911128087599
```

图 1.4.1 kNN 实验结果

1.4.4 决策树算法

实验过程中主要使用 sklearn 函数包来实现决策树方法。决策树（decision tree）算法

基于特征属性进行分类，其主要的优点：模型具有可读性，计算量小，分类速度快。sklearn 决策树算法类库内部实现是使用了调优过的 CART 树算法，既可以做分类，又可以做回归。分类决策树的类对应的是 DecisionTreeClassifier。实现代码如下所示：

```
# 构建 Keras 模型
dtc = DecisionTreeClassifier()
# 训练模式
dtc.fit(partial_train_data, partial_train_targets)
```

首先取十分之一的数据作为测试数据，分别赋給 val_data 与 val_targets；而后准备训练数据，分别赋給 partial_train_data 与 partial_train_targets；之后从 sklearn.tree 包中导入 DecisionTreeClassifier 函数，并初始化；接着以 partial_train_data 与 partial_train_targets 为参数对 DecisionTreeClassifier 进行训练；最后完成精确度与折验证分数平均等指标的统计后。决策树方法的实验结果如图 1.4.2 所示。

```
DecisionTree accuracy:      0.8387096774193549
DecisionTree precision:     0.8970970206264324
DecisionTree recall:        0.8907563025210083
DecisionTree f1 score:      0.8863786540257128
```

图 1.4.2 决策树方法实验结果

1.4.5 多层感知器分类器（MLP Classifier）

实验过程中主要使用 sklearn 函数包来实现多层感知器分类器（MLP Classifier）。MLP 是常见的 ANN（Artificial Neuro Network，人工神经网络）算法，它由一个输入层，一个输出层和一个或多个隐藏层组成。在 MLP 中的所有神经元都差不多，每个神经元都有几个输入（连接前一层）神经元和输出（连接后一层）神经元，该神经元会将相同值传递给与之相连的多个输出神经元，如图所示。本次实验使用 Python 所实现 MLP 算法如下所示：

```
# 构建 Keras 模型
mlp = MLPClassifier(random_state=seed, solver='lbfgs')
# 训练模式
mlp.fit(partial_train_data, partial_train_targets)
```

首先取十分之一的数据作为测试数据，分别赋給 val_data 与 val_targets；而后准备训练数据，分别赋給 partial_train_data 与 partial_train_targets；之后从 sklearn.neural_network 包中导入 MLPClassifier 函数，并初始化；接着以 partial_train_data 与 partial_train_targets

为参数对 MLPClassifier 进行训练；最后完成精确度与折验证分数平均等指标的统计后。多层感知器分类器的实验结果如图 1.4.3 所示。

```
MLPClassifier accuracy:      0.8548387096774194
MLPClassifier precision:     0.9327731092436975
MLPClassifier recall:        0.9124649859943978
MLPClassifier f1 score:      0.9080991672687174
```

图 1.4.3 多层感知器分类器实验结果

1.4.6 朴素贝叶斯算法

实验过程中主要使用 sklearn 函数包来实现朴素贝叶斯方法（Naive Bayes）。朴素贝叶斯（Naive Bayes）属于监督学习的生成模型，实现简单，没有迭代，学习效率高，在大样本量下会有较好的表现。但因为假设太强——假设特征条件独立，在输入向量的特征条件有关联的场景下并不适用。朴素贝叶斯分类器的主要思路：通过联合概率 P(x,y)=P(x|y)P(y)建模，运用贝叶斯定理求解后验概率 P(y|x)；将后验概率最大者对应的的类别作为预测类别。本次实验使用 Python 所实现朴素贝叶斯方法如下所示：

```
# 构建 Keras 模型
nb = GaussianNB()
# 训练模式
nb.fit(partial_train_data, partial_train_targets)
```

首先取十分之一的数据作为测试数据，分别赋給 val_data 与 val_targets；而后准备训练数据，分别赋給 partial_train_data 与 partial_train_targets；之后从 sklearn.naive_bayes 包中导入 GaussianNB 函数，并初始化；接着以 partial_train_data 与 partial_train_targets 为参数对 GaussianNB 进行训练；最后完成精确度与折验证分数平均等指标的统计后。朴素贝叶斯分类器的实验结果如图 1.4.4 所示。

```
NaiveBayes accuracy:     0.8064516129032258
NaiveBayes precision:    0.7620370370370371
NaiveBayes recall:       0.7764550264550264
NaiveBayes f1 score:     0.7399429073342118
```

图 1.4.4 朴素贝叶斯分类器实验结果

1.4.7 SVM 算法

实验过程中主要使用 sklearn 函数包来实现 SVM 方法。SVM 学习的基本想法是求解

能够正确划分训练数据集并且几何间隔最大的分离超平面。本实验借助 sklearn 中的 SVC 函数实现了 SVM 方法，SVC 全称为 C-Support Vector Classification，即支持向量分类，是一种基于 libsvm 实现的，时间复杂度为样本数量平方的算法。Python 实现代码如下所示：

```
# 构建 Keras 模型
svc = SVC(kernel='poly', gamma="auto")
# 训练模式
svc.fit(partial_train_data, partial_train_targets)
```

首先取十分之一的数据作为测试数据，分别赋給 val_data 与 val_targets；而后准备训练数据，分别赋給 partial_train_data 与 partial_train_targets；之后从 sklearn.svm 包中导入 SVC 函数，并初始化；接着以 partial_train_data 与 partial_train_targets 为参数对 SVC 进行训练；最后完成精确度与折验证分数平均等指标的统计后。SVM 的实验结果如图 1.4.5 所示。

```
SVM accuracy:    0.7741935483870968
SVM precision:   0.8781512605042017
SVM recall:      0.8088235294117647
SVM f1 score:    0.8086777044914076
```

图 1.4.5 SVM 实验结果

1.4.8 随机森林算法

实验过程中主要使用 sklearn 函数包来实现随机森林方法。随机森林是一种有监督学习算法，是以决策树为基学习器的集成学习算法。随机森林非常简单，易于实现，计算开销也很小，但是它在分类和回归上表现出非常惊人的性能。Python 实现代码如下所示：

```
# 构建 Keras 模型
rf = RandomForestClassifier(n_estimators=10, max_depth=10)
# 训练模式
rf.fit(partial_train_data, partial_train_targets)
```

首先取十分之一的数据作为测试数据，分别赋給 val_data 与 val_targets；而后准备训练数据，分别赋給 partial_train_data 与 partial_train_targets；之后从 sklearn.ensemble 包中导入 RandomForestClassifier 函数，并初始化；接着以 partial_train_data 与 partial_train_targets 为参数对 RandomForestClassifier 进行训练；最后完成精确度与折验证分数平均等指标的统计后。随机森林算法的实验结果如图 1.4.6 所示。

```
RandomForestClassifier accuracy:    0.8709677419354839
RandomForestClassifier precision:   0.8994708994708994
RandomForestClassifier recall:      0.8789682539682540
RandomForestClassifier f1 score:    0.8760080075869550
```

图 1.4.6 随机森林算法实验结果

1.4.9 bagging 算法

实验过程中主要使用 sklearn 函数包来实现 bagging 算法。bagging 算法的基本流程是：从原始数据中随机抽取 n 个样本，重复 s 次，于是就有个 s 个训练集，每个训练集都可以训练出一个分类器，最终生成 s 个分类器，预测结果将有这些分类器投票决定（选择分类器投票结果中最多的类别作为最后预测结果）。Python 实现代码如下所示：

```
# 构建 Keras 模型
clfb = BaggingClassifier(base_estimator=DecisionTreeClassifier()
                         , max_samples=0.5, max_features=0.5)
# 训练模式
clfb.fit(partial_train_data, partial_train_targets)
```

首先取十分之一的数据作为测试数据，分别赋給 val_data 与 val_targets；而后准备训练数据，分别赋給 partial_train_data 与 partial_train_targets；之后从 sklearn.ensemble 包中导入 BaggingClassifier 函数，并初始化；接着以 partial_train_data 与 partial_train_targets 为参数对 BaggingClassifier 进行训练；最后完成精确度与折验证分数平均等指标的统计后。bagging 算法的实验结果如图 1.4.7 所示。

```
Bagging accuracy:    0.8548387096774194
Bagging precision:   0.9411764705882353
Bagging recall:      0.9257703081232492
Bagging f1 score:    0.9169367268438476
```

图 1.4.7 bagging 算法实验结果

1.5 实验分析

1.5.1 算法最佳性能对比

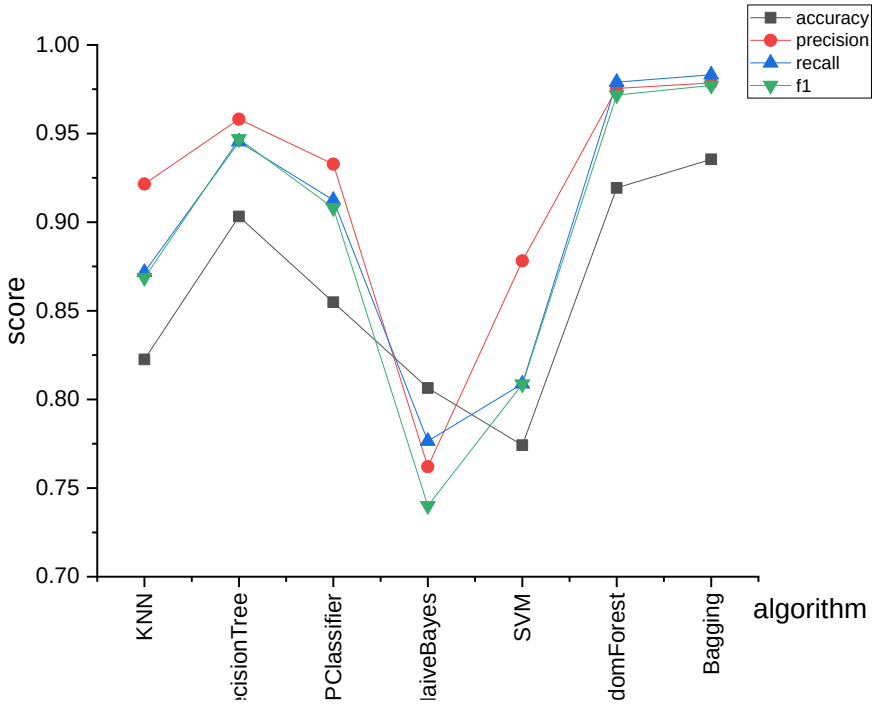

图 1.5.1 算法最佳指标对比

本次实验计算了各个模型在 UCI soybean 数据集上的最佳性能，一共运行了 20 次程序，最终所得各模型的最佳的准确率、精确率、召回率与 f1 值如图 1.5.1 所示。

从实验结果可知，对于 UCI soybean 数据集，本次实验所实现的各个模型的最佳性能结果如下：准确率最佳的为 bagging 算法，其后的依次是随机森林算法、决策树算法、多层感知器分类器、kNN 算法、朴素贝叶斯、SVM，显然准确率最低的是 SVM；精确度最佳的为 bagging 算法，其后的依次是随机森林算法、决策树算法、多层感知器分类器、kNN 算法、SVM、朴素贝叶斯，显然准确率最低的是朴素贝叶斯；召回率最佳的为 bagging 算法，其后的依次是随机森林算法、决策树算法、多层感知器分类器、kNN 算法、SVM、朴素贝叶斯，召回率最低的也是朴素贝叶斯；f1 率最佳的为 bagging 算法，其后的依次是随机森林算法、决策树算法、多层感知器分类器、kNN 算法、SVM、朴素贝叶斯，f1 率最低的仍是朴素贝叶斯。

1.5.2 算法平均性能对比

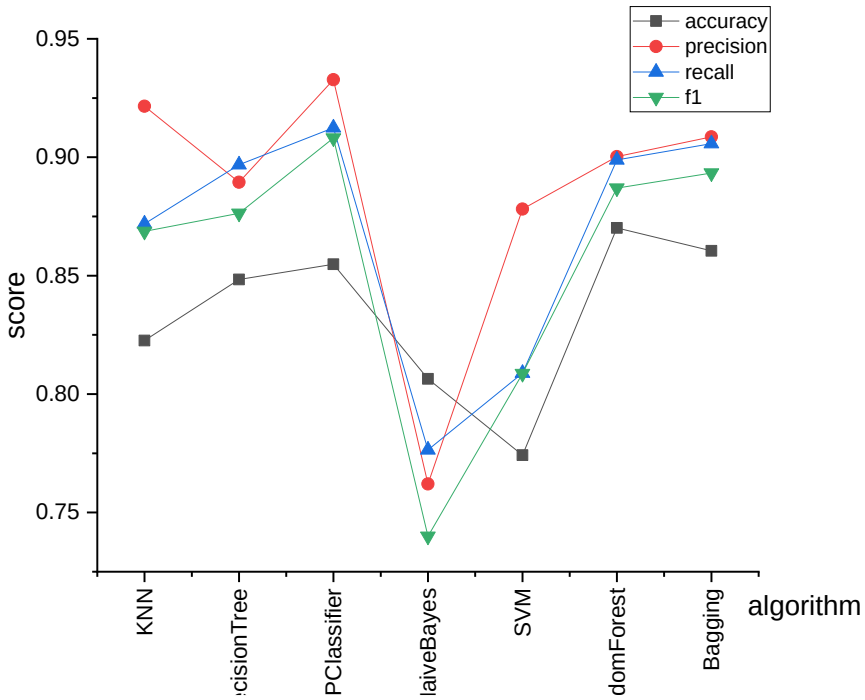

图 1.5.2 算法平均指标对比

本次实验计算了各个模型在 UCI soybean 数据集上的平均性能，一共运行了 20 次程序，最终所得各模型的平均的准确率、精确率、召回率与 f1 值如图 1.5.2 所示。

从实验结果可知，对于 UCI soybean 数据集，本次实验所实现的各个模型的平均性能结果如下：准确率最佳的为随机森林算法、其后的依次是 bagging 算法、多层感知器分类器、决策树算法、kNN 算法、朴素贝叶斯、SVM 算法，显然准确率最低的是 SVM；精确度最佳的为多层感知器分类器，其后的依次为 kNN 算法、bagging 算法、随机森林算法、决策树算法、SVM 算法、朴素贝叶斯，显然准确率最低的是朴素贝叶斯；召回率最佳的为多层感知器分类器，其后的依次为 bagging 算法、随机森林算法、决策树算法、kNN、SVM 算法、朴素贝叶斯，显然召回率最低的也是朴素贝叶斯；f1 率最佳的为多层感知器分类器，其后的依次为 bagging 算法、随机森林算法、决策树算法 kNN、SVM 算法、朴素贝叶斯，显然 f1 率最低的仍是朴素贝叶斯。

1.5.3 测试样例占比对算法结果的影响

本次实验以随机森林算法为例。

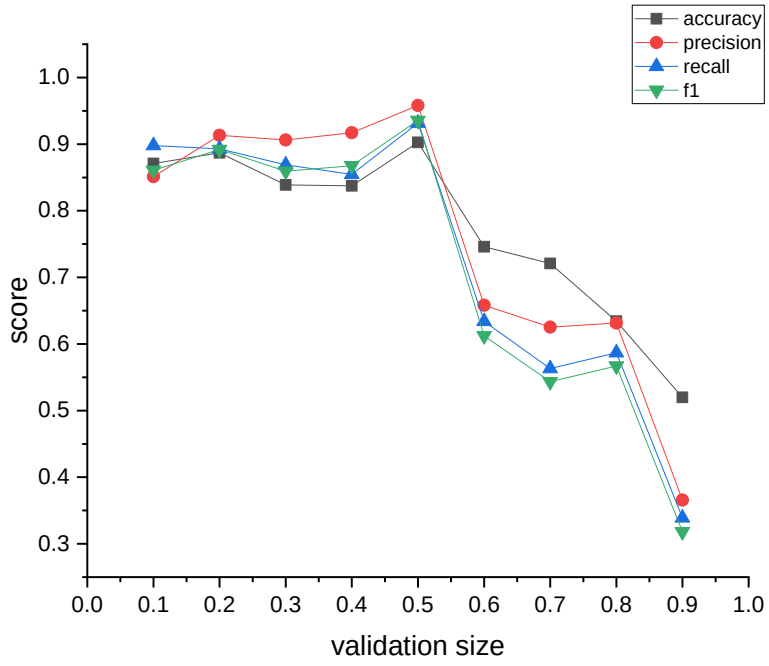

图 1.5.3 随机森林算法性能指标与测试样例占比关系

本次实验计算了随机森林算法的性能在 UCI soybean 数据集上的随验证样例占比的变化，最终所得的准确率、精确率、召回率与 f1 值的改变趋势如图 1.5.3 所示。

从实验结果可知，对于 UCI soybean 数据集，各项指标与验证样例占比的关系大致为：验证样例占比从 0.1 增加到 0.2 时，除了召回率略有下降外，其他三项指标均有所上升；验证样例占比从 0.2 增加到 0.4 时，除了准确率下降明显外，其他三项指标虽略有下降但总体平稳；验证样例从 0.4 增加到 0.5 时，四项指标均有上升，其中准确率上升显著，近 7%；验证样例从 0.5 增加到 0.7 时，四项指标均大幅度下降，0.5 至 0.6 段尤其显著；**值得一提的是，准确率以 0.5 为分水岭，之后一直下降，未见上升**；验证样例从 0.7 增加到 0.8 时，除准确率外，其余三项指标均小幅度上升；验证样例从 0.8 增加到 0.9 时，各项指标均大幅度下降。

1.5.4 初始化随机数对算法结果的影响

本次实验以 MLP 算法为例。

本次实验计算了 MLP 算法的性能在 UCI soybean 数据集上的随初始化随机数的变化，最终所得的准确率、精确率、召回率与 f1 值的改变趋势如图 1.5.4 所示。

从实验结果可知，对于 UCI soybean 数据集，各项指标与初始化随机数的关系大致为：初始化随机数从 1 增长至 2 时，各项指标均上升明显，超过 20%；初始化随机数从 2 增长至 3 时，各项指标均有明显程度的下降；其中准确率下降较小；初始化随机数从 3 增长至 4 时，除准确率大致持平外，其余三项指标均有 3%~5%的上升；初始化随机数从 4 增长至 5 时，各项指标均有超过明显的下降，其中召回率下降达 13%；初始化随机数从 5 增长至 6 时，各项指标均上升，其中准确率上升程度为 8.1%，精确率上升程度是 19.5%，召回率与 f1 上升几乎持平：21.1%与 21.2%；初始化随机数上升到 6 以后，各项性能指标均一直在显著下降。

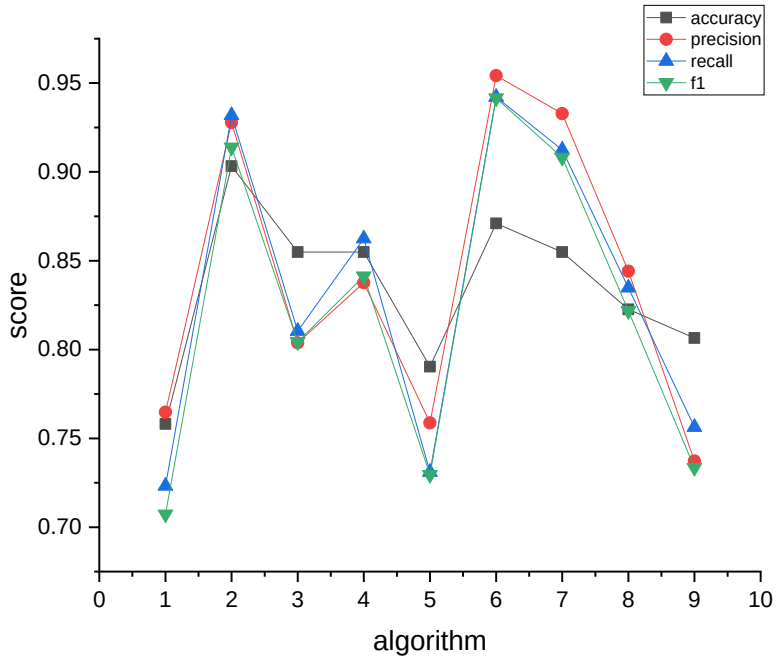

图 1.5.4 MLP 算法性能指标与初始化随机数关系

1.6 实验总结

经过这次实验的训练与实践，我认识到了数据集对于算法性能好坏的重要性，一个好的数据集不仅可以最大程度地挖掘出算法的潜力，还可以帮助算法设计者改正算法的漏洞。值得一提的是，数据集的收集与整理是极其耗费人力与物力的一项工作；

除此以外，我还体验了数据预处理的步骤，自己动手实现了数据预处理的完整流程，加深了对课本知识的认知；

本次实验最大的体会莫过于进一步加深了对于若干个经典算法的理解，如：kNN 算法、决策树算法、多层感知器分类器、朴素贝叶斯算法、SVM 算法、随机森林算法与 bagging 算法等。学会借助 sklearn 包实现各个算法的训练与测试。在不断的尝试中，个人的参数调节能力得到了强化与提高，从不同算法对 soybean 的性能比较，直观地了解其内在差异，明白了如何根据具体数据集的数据特点与分布选择合适的算法。

附录

验收时老师提问与个人回答

问：你怎么实现决策树算法的？

答：主要使用 Python 的第三方工具 sklearn 函数包来实现决策树方法，sklearn 有许多现成的工具可以调用，避免了每一次都要手动“造轮子”的尴尬，使用起来很方便。

问：你怎么调节参数？为什么？

答：通过理论学习了解了一些参数的普遍默认值，同时在实验过程中不断实践，直到找到能够取得最佳性能的参数。

2 实验 2：基于 UCI Groceries Dataset 的关联分析任务

2.1 背景

近年来我国的零售行业发展迅速，越来越多的越多的大型企业或超市开始将目光投向数据挖掘技术，有效的利用数据挖掘技术为企业提供信息是各大零售巨头核心竞争力的重要组成部分。通过查看哪些商品经常一起购买，可以帮助商家了解用户的购买行为。这种从数据的海洋中抽取的知识可以用于商品的定价、市场促销、存货管理等环节。其重要技术就是利用关联分析挖掘潜在用户，提高销售额。从大规模的数据集中寻找物品间的隐含关系被称作关联分析或者关联规则学习。这里的主要问题在于，寻找物品的不同组合是一项十分耗时的任务，所需的计算代价很高，蛮力搜索方法并不能解决这个问题，所以需要用更智能的方法在合理的时间范围内找到频繁项集。

传统的关联规则的缺点就是没有考虑关联规则的商业价值。如果采用传统的关联规则评价标准，同时销售“一瓶昂贵的红酒和一盒鱼子酱”与销售“一盒牛奶和一袋面包”对于关联规则挖掘过程来说意义相差不大。但事实上，零售商会更倾向推销前者，显然该组合能为企业带来比后者更大的净利润。利用开放出来的超市购物蓝数据进行关联规则分析，因为人们在选择购买商品的时候往往会购买多个商品，如果这些商品摆放在一起，就会增加用户购买的可能性。

本实验在关联分析 Apriori 经典算法的基础上，充分考虑了利润收益，提出了运用频繁项目集中各个项目的加权利润之和来表示交叉销售的利润的思想评估关联规则价值，并且在 UCI Groceries 销售数据上进行了频繁项集的挖掘，提出各种捆绑销售模型，具有很强的实用价值。

2.2 问题描述

本实验解决的问题归纳如下：在已知的销售数据集，包括如商品 ID；商品名称，价格；销售时间等信息的基础上，进行关联性分析，挖掘到频繁项目集，给出频繁项目集的直方图，从而进行合理的产品摆放和个性化推荐，从而提高销售利润。

频繁项目集通常表示顾客一次购物时同时购买的商品组合，因此评估一项商品的商业价值时，除了考虑该商品本身所产生的独立利润外，同时也应考虑由于交叉销售时带来的额外利润。

2.3 实验环境

- **硬件：**Dell G3 3579 笔记本
- 软件：

  OS：Windows 10 Pro N for Workstations

  平台工具：PyCharm 2019.3.4 (Professional Edition)、Python 3.7.4、OriginPro 2018(64-bit)

2.4 数据集及实现的技术方案

为了实现频繁项集的挖掘，本文利用 Apriori 算法来寻找不同的商品组合。图 2.4.1 展示了本实验方案处理的流程。

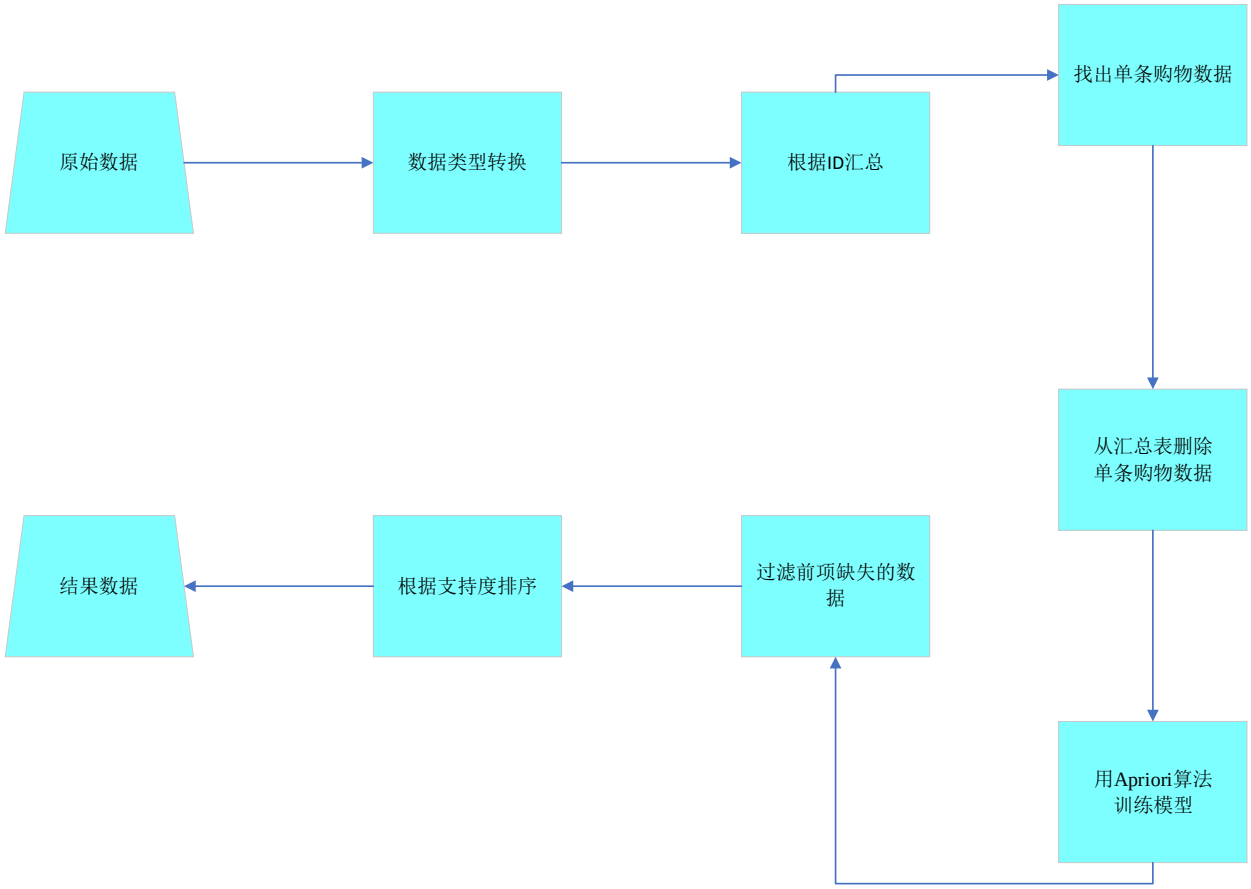

图 2.4.1 技术方案流程图

2.4.1 数据集介绍及预处理

本实验选择 UCI Groceries 销售数据作为数据挖掘样本。共 9836 条数据，其中包括了商品名称、购买 ID，其数据格式如表 2.4.1 所示。

表 2.4.1 数据集样本数据格式

id	items
1	{citrus fruit,semi-finished bread,margarine,ready soups}
2	{tropical fruit,yogurt,coffee}
3	{whole milk}
4	{pip fruit,yogurt,cream cheese ,meat spreads}
5	{other vegetables,whole milk,condensed milk,long life bakery product}
6	{whole milk,butter,yogurt,rice,abrasive cleaner}
7	{rolls/buns}
8	{other vegetables,UHT-milk,rolls/buns,bottled beer,liquor (appetizer)}
9	{pot plants}
10	{whole milk,cereals}
11	{tropical fruit,other vegetables,white bread,bottled water,chocolate}
12	{citrus fruit,tropical fruit,whole milk,butter,curd,yogurt,flour,bottled water,dishes}
13	{beef}
14	{frankfurter,rolls/buns,soda}
15	{chicken,tropical fruit}
16	{butter,sugar,fruit/vegetable juice,newspapers}
17	{fruit/vegetable juice}

为了达到更好的实验效果，在频繁项集的挖掘之前需要对原始数据集进行预处理，考虑采用的方法包括：

（1）将商品中的 ID 列剔除；

（2）对于同一件商品由于口味的不同导致的商品名字不同的情况，将商品名均用其中某一个名称表示（如牛奶）；

（3）对于同一件商品由于尺寸的不同导致的商品名字不同的情况，将商品名均用其中某一个名称表示。

2.4.2 频繁项集挖掘

考虑数据集特点和计算硬件条件，决定采用 Apriori 算法实现，而 Apriori 算法的基本思想是首先是找出所有大于最小支持度的频繁项集，然后由频繁项集产生关联规则，这些规则必须满足最小支持度和最小可信度。Apriori 算法是用来发现频繁项集的一种方法。Apriori 算法的两个输入参数分别是最小支持度和数据集。该算法首先生成所有单个物品的项集列表，遍历之后去掉不满足最小支持度要求的项集；接下来对剩下的集合进行组合生成包含两个元素的项集，去掉不满足最小支持度的项集；重复该过程直到去掉所有

不满足最小支持度的项集。Apriori 算法流程如图 2.4.2 所示。

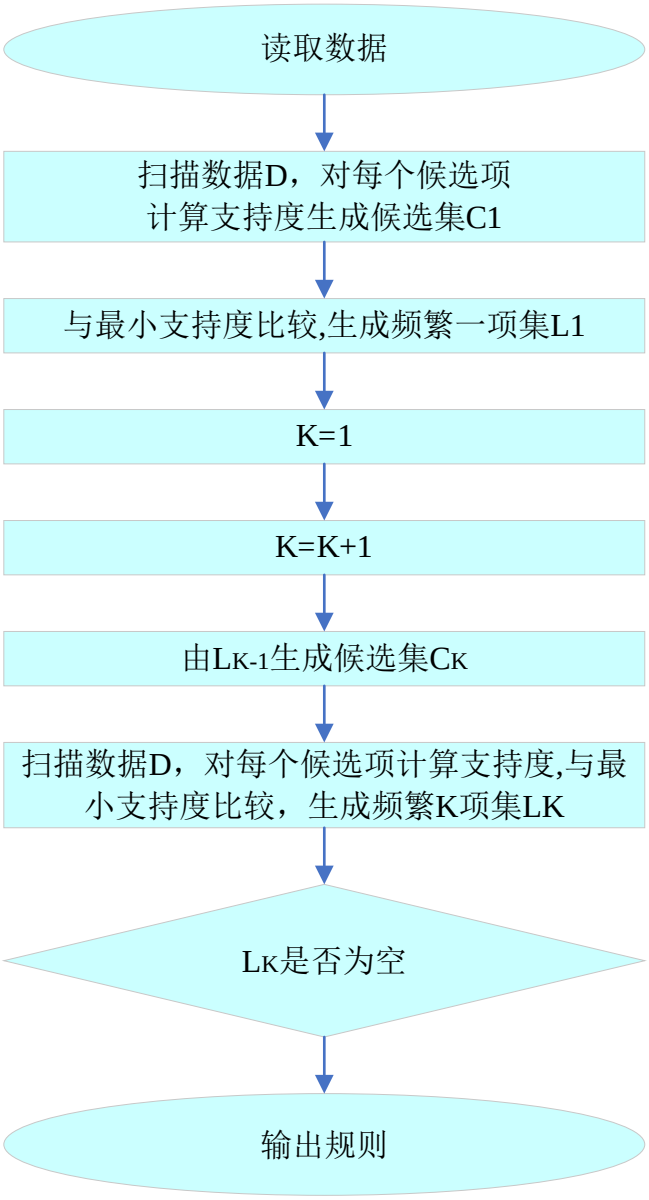

图 2.4.2 Apriori 算法流程

Apriori 算法实现的伪代码如下:

输入：
D：事务数据库；
min_sup：最小支持度计数阈值。
输出：L：D 中的频繁项集。
方法：
1) L1 = find_frequent_1_itemsets(D);
2) for (k = 2; $L_{k-1} \neq \varnothing$; k++) {

```
3)              Ck = aproiri_gen(Lk-1,min_sup);
4)              for each transaction t∈D{ //扫描 D 用来计数
5)                      Ct = subset(Ck,t); //找出事务 t 中包含的所有候选 k 项集，
6)                      for each candidate c∈Ct //对事务 t 包含的每个候选 k 项集的计数加一
7)                              c.count++;
8)              }
9)              Lk={c∈Ck | c.count ≥ min_sup}
10)     }
11)     return L = ∪kLk；
procedure apriori_gen(Lk-1: frequent (k-1)-itemset; min_sup: support)
1)      for each itemset l1∈Lk-1
2)              for each itemset l2∈Lk-1
3)                      if (l1[1]=l2[1])∧...∧(l1[k-2]=l2[k-2])∧(l1[k-1]<l2[k-2]) then {
4)                              c = l1 连接 l2; //连接步: 产生 candidates
5)                              if has_infrequent_subset(c,Lk-1) then
6)                                      delete c; // 剪枝步: 移除非频繁的 cadidate
7)                              else add c to Ck;
8)                      }
9)      return Ck;
procedure has_infrequent_subset(c:candidate k-itemset; Lk-1:frequent
(k-1)-itemset)
//使用先验知识
1)      for each (k-1)-subset s of c
2)              if c∉Lk-1 then
3)                      return TRUE;
4)      return FALSE;
其中， Lk-1 表示频繁 k-1 项集。
```

2.4.3 频繁项目集挖掘

在所得到的关联规则中，留下符合最小支持度和最小置信度的规则，计算出每一条关联规则的所对应的利润。设计优化的目标函数（本实验中所采用的目标函数为$J = \alpha \cdot \sup port + profit$），在最小支持度与利润（原始数据集没有給出利润，故本实验虚拟了若干利润值）之间做出权衡。

2.5 实验结果

本实验首先对数据集进行预处理，将同一件商品由于口味或者尺寸不同而导致的商品名称的差异进行一般化。并将商品中的空余项删除。然后，通过商品 ID 将销售数据合并成购物篮元组数据，为下一步的关联规则挖掘做准备。

利用 Apriori 算法进行频繁项集的挖掘，并且根据所挖掘到的频繁项集自动生成关联规则。本实验中设置最小支持度 minSupport=0.015，最小置信度 minConf=0.1。所挖掘得到的频繁项集的直方图如图 2.5.1 所示。

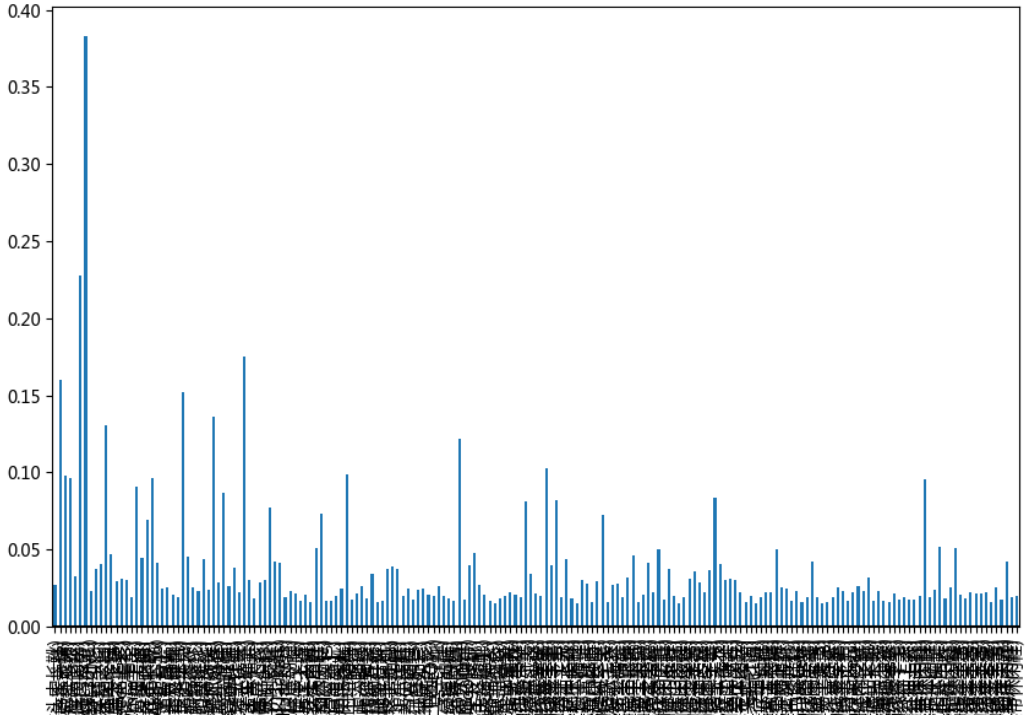

图 2.5.1 关联规则直方图

尽管最小支持度和最小置信度阈值有助于排出大量无趣规则的探查，但仍然会产生一些用户不感兴趣的规则，故本实验中采用提升度（lift）来作为相关性的度量。如果提升度的值小于 1，则表示关联规则的前项和后项是负相关的，如果值为 1，则前项和后项是相互独立的，如果值大于 1，意味着前项和后项是正相关的。计算出每一条规则对应的提升度，并且根据利润表，查询得到每一条关联规则各商品加权的利润，将其一起写入关联规则中。最终得到的部分关联规则如表 2.5.1 所示.

表 2.5.1 部分关联规则

from	to	support	conf	lift
baking powder	other vegetables	0.007320793	0.413793103	2.138547122
baking powder	whole milk	0.009252669	0.522988506	2.046793456
beef	other vegetables	0.01972547	0.375968992	1.943066232
beef	rolls/buns	0.013624809	0.259689922	1.411857594
beef	root vegetables	0.017386884	0.331395349	3.040366843

beef	whole milk	0.021250635	0.40503876	1.585179547
berries	other vegetables	0.010269446	0.308868502	1.596280459
berries	whipped/sour cream	0.009049314	0.272171254	3.796885505
berries	whole milk	0.011794611	0.354740061	1.388328095
berries	yogurt	0.010574479	0.318042813	2.279847719
beverages	whole milk	0.006812405	0.26171875	1.024275331
bottled water	soda	0.028978139	0.262189512	1.503576592
bottled water	whole milk	0.034367056	0.310947562	1.216939623
brown bread	other vegetables	0.018708693	0.288401254	1.490502539
brown bread	whole milk	0.025216065	0.388714734	1.521293038

对于所得到的关联规则，将关联规则前项和后项是负相关或者独立的规则剔除（即提升度 lift<=1 的规则），选取优化函数$J = \alpha \cdot \sup port + \beta \cdot profit$，此处可以将参数 β 设为 1，优化函数 J 不变。此时优化目标函数为$J = \alpha \cdot \sup port + profit$，在本实验中，选取参数 α 为 100，计算每一条规则对应的 J，根据 J 的大下对关联规则进行排列，如表 2.5.2 所示。最后根据需求决定采用哪些关联规则应用于商家的营销决策中。

表 2.5.2 优化后的部分关联规则

from	to	support	conf	lift	profit	target
long life bakery product	other vegetables	0.010676157	0.285326087	1.474609598	164	165.84
long life bakery product	whole milk	0.013523132	0.361413043	1.414443805	200	205.61
margarine	other vegetables	0.01972547	0.336805556	1.740663499	63.1	64.217
margarine	rolls/buns	0.014743264	0.251736111	1.368615065	148	150.15
margarine	whole milk	0.024199288	0.413194444	1.617098035	180	182.68
meat	other vegetables	0.009964413	0.385826772	1.994012769	210	211.89
meat	rolls/buns	0.006914082	0.267716535	1.455495924	173	176.91
meat	whole milk	0.009964413	0.385826772	1.509990569	102	104.22
misc. beverages	soda	0.007320793	0.258064516	1.479921001	140	143.78
napkins	other vegetables	0.014438231	0.275728155	1.425005995	256.5	258.17
napkins	whole milk	0.01972547	0.376699029	1.474267788	148.01	149.81
newspapers	whole milk	0.027351296	0.342675159	1.341110303	108	109.2
oil	other vegetables	0.009964413	0.355072464	1.835069722	152.1	153.63
oil	root vegetables	0.00701576	0.25	2.293610075	110	116.73

oil	whole milk	0.011286223	0.402173913	1.573967543	187.6	190.88
onions	other vegetables	0.014234875	0.459016393	2.372268119	40.23	41.976
onions	root vegetables	0.009456024	0.304918033	2.797452288	86.8	85.562
onions	whole milk	0.012099644	0.390163934	1.526964702	150.2	153.01
pastry	other vegetables	0.022572445	0.253714286	1.311234892	184.3	186.4
pickled vegetables	other vegetables	0.006405694	0.357954545	1.849964769	96.3	98.282

对于上列的关联规则，商家可以选择，将“long life bakery product”+“ whole milk”+“margarine”进行捆绑销售，类似地，可以做出一系列的营销决策，并且可以后续的销售情况将参数 α 调节至合适的值，以实现利润的最大化。

附录

问：代码是自己写的吗？

答：是的！参考资料结合实验需要改写得到的。

3　实验 3：基于 PACS RAW Labeled Dataset 的聚类任务

3.1　实验目的

通过研究 PACS RAW 标签数据集分布与结构的差异化，可以更深刻的掌握聚类分析的原理；进一步熟悉聚类分析问题的提出、解决问题的思路、方法和技能；达到能综合运用所学基本理论和专业知识；锻炼收集、整理、运用资料的能力的目的；能使用 Python 语言编写相应代码进行聚类实验操作，并且可以对数据处理结果进行正确判断分析，作出综合评价。

3.2　实验任务

根据聚类分析的原理，借助 DBSCAN 聚类方法对給定数据进行聚类分析，給出并解释实验结果。

3.3　实验环境

- **硬件：**Dell G3 3579 笔记本
- 软件：

  OS：Windows 10 Pro N for Workstations

  平台工具：PyCharm 2019.3.4 (Professional Edition)、Python 3.7.4、OriginPro 2018(64-bit)

3.4　实验内容

3.4.1　库函数引用

本实验引用第三方函数工具包主要是 numpy、math 与 matplotlib。

```
import numpy as np

from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets.samples_generator import make_blobs
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import math
```

3.4.2　密度聚类（DBSCAN）

DBSCAN 的全称是 Density-Based Spatial Clustering of Applications with Noise，中文含义是“基于密度的带有噪声的空间聚类”，密度聚类的思想是不同于 K-Means 的，但是更符合我们人类的思维，基本的思想是通过是否紧密相连来判断样本点是否属于一个簇。

代表性的算法就是 DBSCAN，它基于一组邻域参数(ϵ,MinPts)来表征某处样本是否是紧密的。

3.4.3 伪代码与流程图

DBSCAN 算法的伪代码描述如下所示。

```
DBSCAN(dataset, eps, MinPts){
# cluster index
C = 1
for each unvisited point p in dataset {
        mark p as visited
        # find neighbors
        Neighbors N = find the neighboring points of p

        if |N|>=MinPts:
            N = N U N'
            if p' is not a member of any cluster:
                add p' to cluster C
}
```

DBSCAN 算法的流程图如图所示。

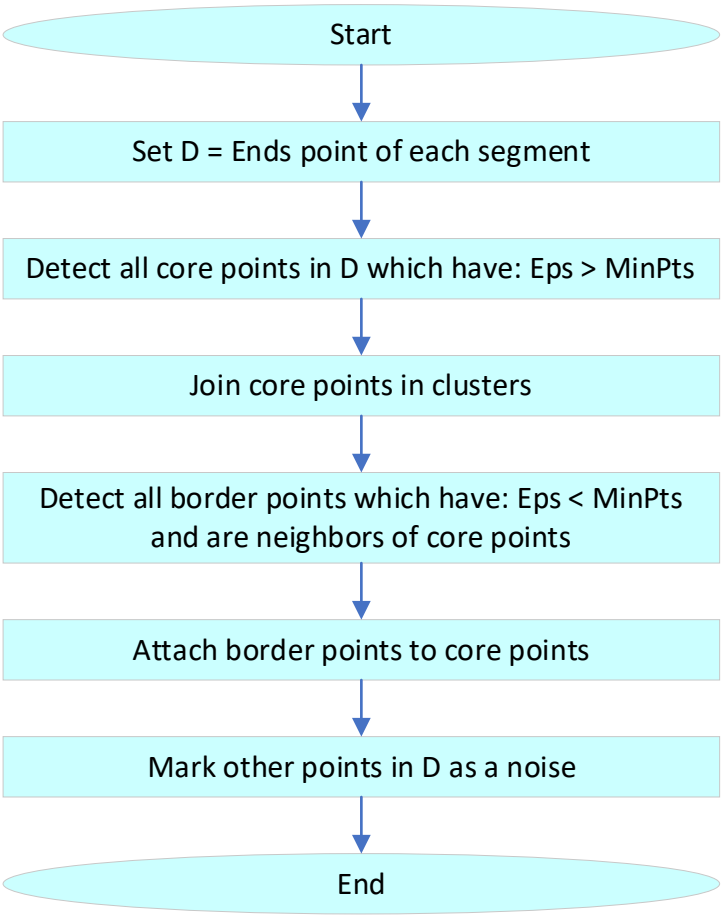

图 3.4.1 基本 DBSCAN 算法流程图

3.4.4 核心代码

本次实验使用 Python 所实现 DBSCAN 代码如下所示。

```
def dbscan(X, eps, minpts):
  rows = X.shape[0]
  classification_result = [INITIAL] * rows
  core_point_list = list()
  cluster_id = 0
  for i in range(rows):
    if classification_result[i] == INITIAL:
      neighborhoods  = pointsWithinEps(X[i],X,eps)
      if len(neighborhoods) < minpts - 1:
        classification_result[i] = NOISE
      else:
        # find the core points
        core_point_list.append(i)
        classification_result[i] = cluster_id
        unClassification_neighborhoods = []
        for item in neighborhoods:
```

```
            # this check to make sure already classified points do not classified again
            if classification_result[item] == INITIAL or classification_result[item] == NOISE:
              classification_result[item] = cluster_id
              unClassification_neighborhoods.append(item)
          # find if there have any unclassified core points in the neighbours
          while len(unClassification_neighborhoods) > 0:
            need_check_index = unClassification_neighborhoods[0]
            check_result = pointsWithinEps(X[need_check_index],X,eps)
            if len(check_result) >= minpts:
              core_point_list.append(need_check_index)
              for i in range(len(check_result)):
                point = check_result[i]
                if classification_result[point] == INITIAL or  classification_result[point] ==
NOISE:
                  if classification_result[point] == INITIAL:
                    unClassification_neighborhoods.append(point)
                  classification_result[point] = cluster_id
            unClassification_neighborhoods = unClassification_neighborhoods[1:]
          cluster_id = cluster_id + 1
    core_point_list.sort()
    return [classification_result,core_point_list]
```

3.5 实验分析

如图 3.5.1 至图 3.5.3 所示为本次实验结果。

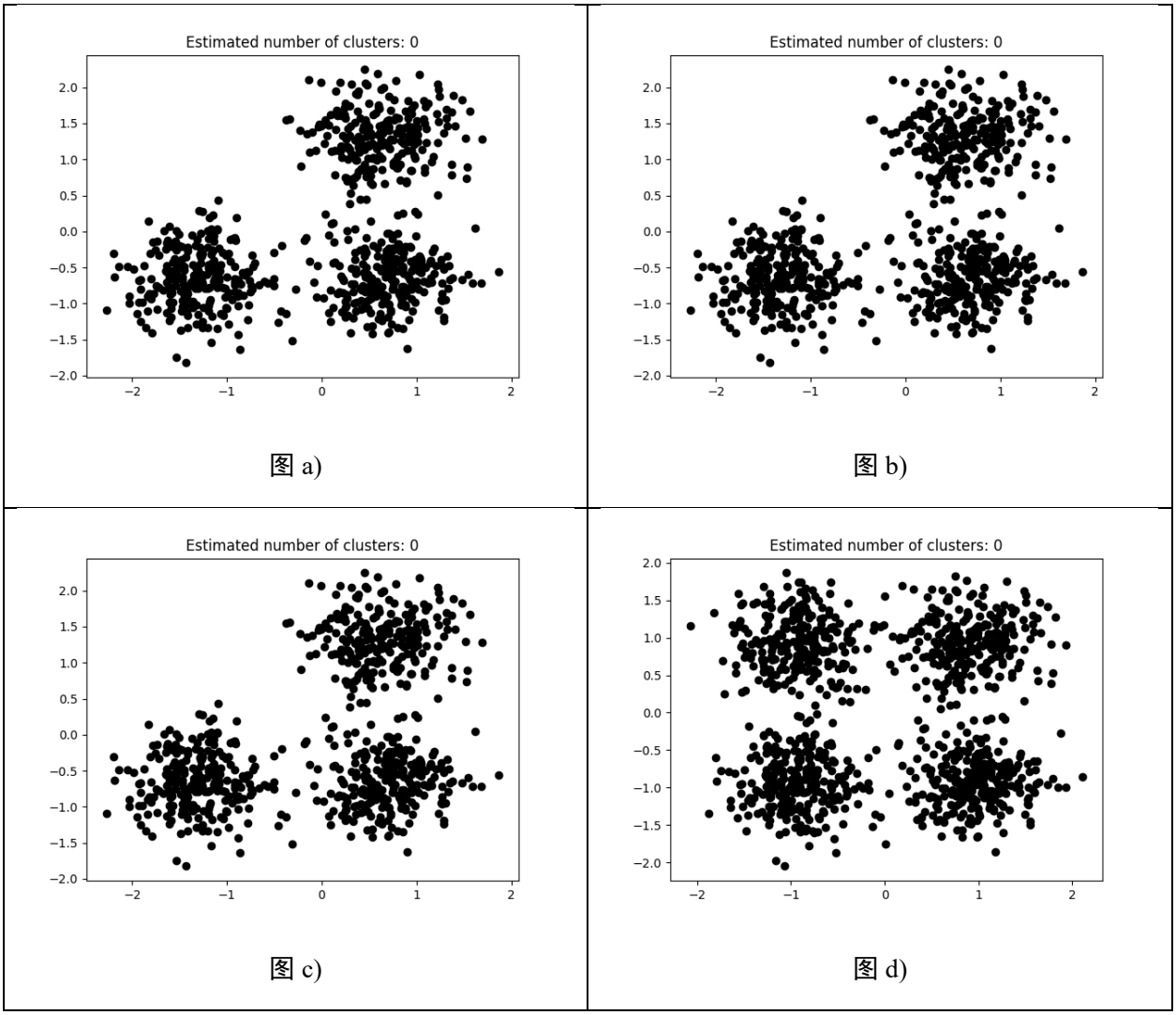

图 3.5.1 聚类结果-1

图 3.5.1 中图 a)至图 d)的 eps 与 minpts 依次分别为：[0.015, 10]、[0.020, 10]、[0.035, 20]与[0.061, 20]。

图 3.5.2 中图 a)至图 d)的 eps 与 minpts 依次分别为：[0.9, 40]、[0.3, 10]、[0.2, 10]与[0.07, 10]。

图 3.5.3 中图 a)至图 d)的 eps 与 minpts 依次分别为：[0.091, 10]、[0.093, 10]、[0.10, 10]与[0.0.096, 10]。

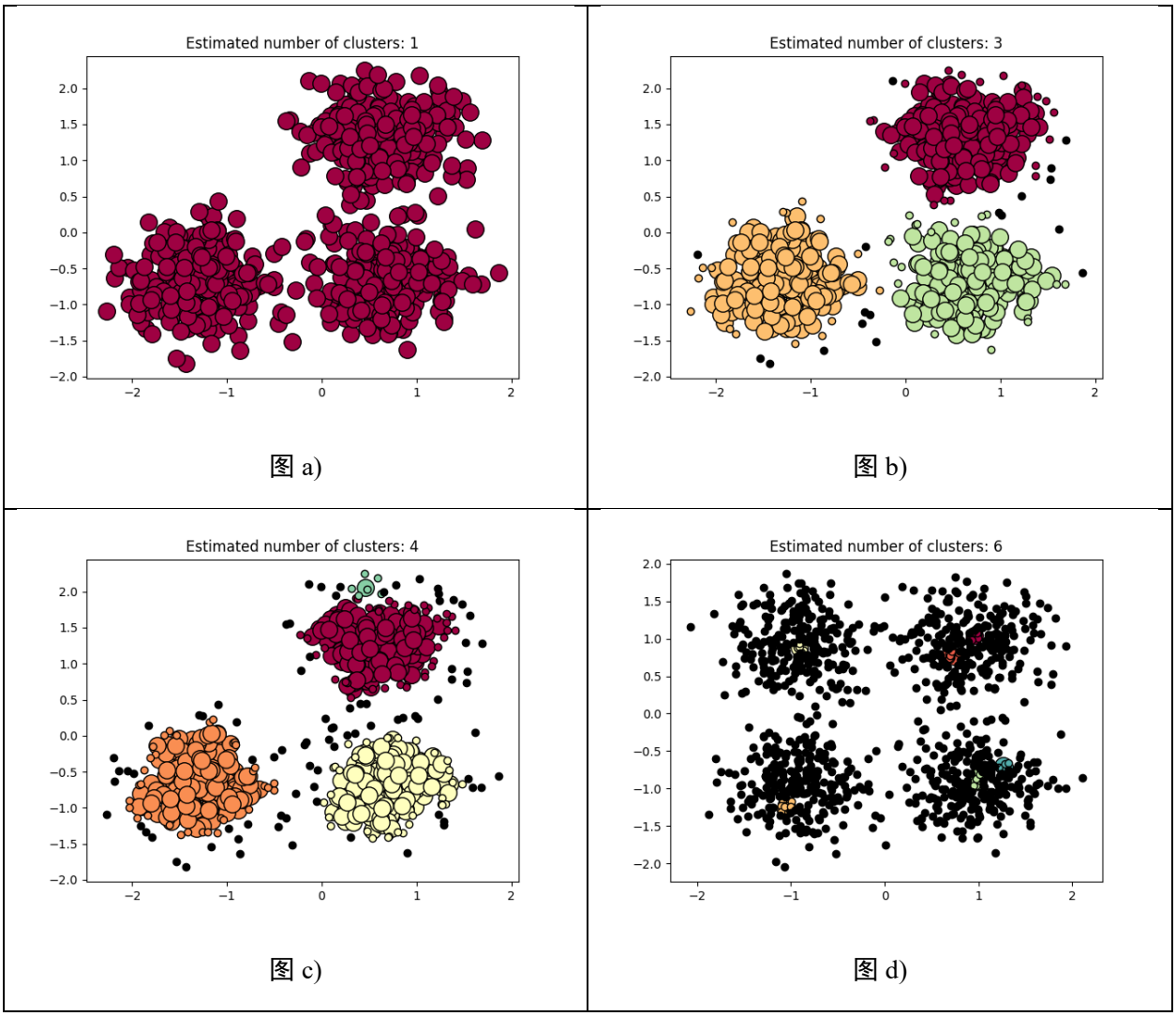

图 a)　　图 b)

图 c)　　图 d)

图 3.5.2 聚类结果-2

表 3.5.1 聚类结果参数与聚类数目

eps	minpts	Estimated number of cluster(s)
0.015	10	0
0.020	10	0
0.035	20	0
0.061	20	0
0.900	40	1
0.300	10	3
0.200	10	4
0.070	10	6
0.091	10	11
0.093	10	12
0.100	10	13
0.096	10	15

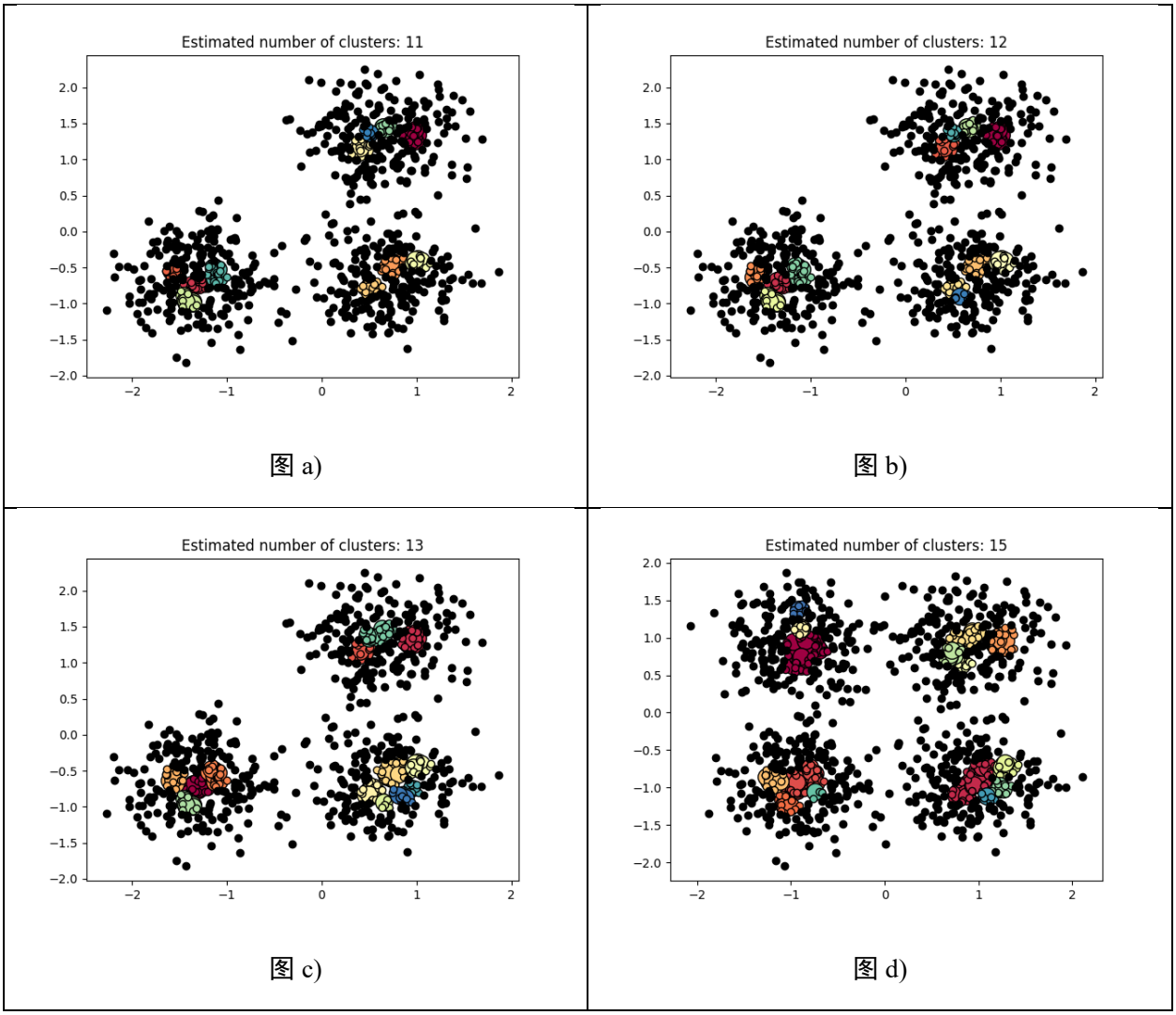

图 a)　　图 b)

图 c)　　图 d)

图 3.5.3 聚类结果-3

表 3.5.1 所示为图 3.5.1 至图 3.5.3 的参数与估计聚类数目汇总，从实验结果可知：minpts 与 eps 的比值过大时，估计的聚类数目一般为 0；随着两者比值减小，估计的聚类数目增大；当比值到达某一阈值时，估计的聚类数目又开始减少。

本次实验所使用的聚类算法是 DBSCAN，与 K-Means 相比，其具有以下优点：

●原始数据分布规律没有明显要求，能适应任意数据集分布形状的空间聚类，因此数据集适用性更广，尤其是对非凸装、圆环形等异性簇分布的识别较好；

●无需指定聚类数量，对结果的先验要求不高；

●由于 DBSCAN 可区分核心对象、边界点和噪点，因此对噪声的过滤效果好，能有效应对数据噪点。

由于他对整个数据集进行操作且聚类时使用了一个全局性的表征密度的参数，因此也存在比较明显的弱点：

●对于高纬度问题，基于半径和密度的定义成问题；

●当簇的密度变化太大时，聚类结果较差；

●当数据量增大时，要求较大的内存支持，I/O 消耗也很大。

附录

问：聚类聚了几类？

答：本次实验我使用的聚类算法是 DBSCAN，每次的实验结果随着 minpts 与 eps 的不同而变化。