湖南科技大学计算机科学与工程学院

数据挖掘 课程设计报告

专业班级： 计科六班

姓 名： 周俊哲

学 号： 2205010711

指导教师： 胡蓉

时 间：

地 点： 逸夫楼430

指导教师评语：

成绩： 等级：

签名：

年 月

1. **实验题目**

**实验一、Apriori算法设计与应用**

1. **背景介绍**

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

1. **实验内容（包括：实验原理/运用的理论知识、算法/程序流程图、步骤和方法、关键源代码）**

**关键源代码：// Apriori 算法核心实现**

**static (HashSet<HashSet<string>>, Dictionary<HashSet<string>, double>) AprioriAlgorithm(List<HashSet<string>> transactions, double minSupport)**

**{**

**int numTransactions = transactions.Count;**

**var supportData = new Dictionary<HashSet<string>, double>(new HashSetComparer());**

**var freqItemsets = new HashSet<HashSet<string>>(new HashSetComparer());**

**// 生成频繁 1 项集**

**var itemCounts = new Dictionary<string, int>();**

**foreach (var transaction in transactions)**

**{**

**foreach (var item in transaction)**

**{**

**if (!itemCounts.ContainsKey(item))**

**itemCounts[item] = 0;**

**itemCounts[item]++;**

**}**

**}**

**// 过滤满足最小支持度的项**

**var currentItemsets = new HashSet<HashSet<string>>(new HashSetComparer());**

**foreach (var item in itemCounts.Keys)**

**{**

**double support = (double)itemCounts[item] / numTransactions;**

**if (support >= minSupport)**

**{**

**var itemset = new HashSet<string> { item };**

**currentItemsets.Add(itemset);**

**supportData[itemset] = support;**

**freqItemsets.Add(itemset);**

**}**

**}**

**// 生成更大的频繁项集**

**int k = 2;**

**while (currentItemsets.Count > 0)**

**{**

**var candidates = GenerateCandidates(currentItemsets, k);**

**var candidateCounts = new Dictionary<HashSet<string>, int>(new HashSetComparer());**

**foreach (var transaction in transactions)**

**{**

**foreach (var candidate in candidates)**

**{**

**if (candidate.IsSubsetOf(transaction))**

**{**

**if (!candidateCounts.ContainsKey(candidate))**

**candidateCounts[candidate] = 0;**

**candidateCounts[candidate]++;**

**}**

**}**

**}**

**currentItemsets.Clear();**

**foreach (var candidate in candidateCounts.Keys)**

**{**

**double support = (double)candidateCounts[candidate] / numTransactions;**

**if (support >= minSupport)**

**{**

**currentItemsets.Add(candidate);**

**supportData[candidate] = support;**

**freqItemsets.Add(candidate);**

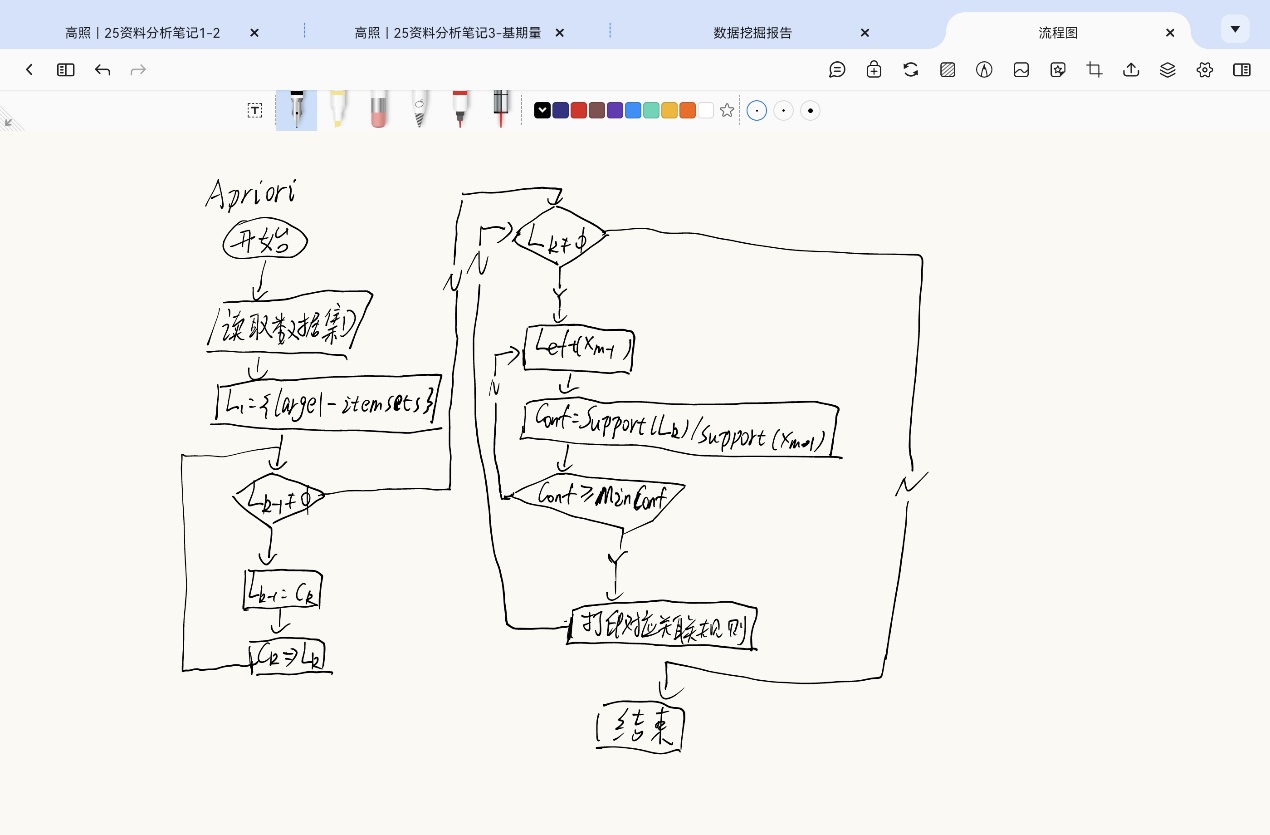
**}**

**}**

**k++;**

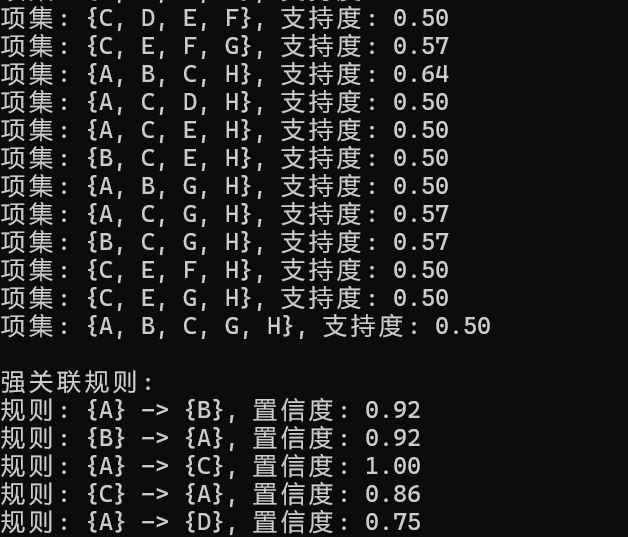
**}**

**return (freqItemsets, supportData);**

**}**

**Apriori算法流程图**

1. **实验结果与分析**

结果如下：

**五、小结与心得体会**

Apriori 算法简单，易于实现。但是它也有自己的缺点，数据集很大的时会 出现下面两个问题。 1. 需要多次扫描数据集 2. 可能会产生庞大的候选集

1. **实验题目**

**实验二 Close算法设计与应用**

1. **背景介绍**

一个频繁闭合项目集的所有闭合子集一定是频繁的；一个非频繁闭合项目集的所有闭合超集一定是非频繁的。

1. **实验内容（包括：实验原理/运用的理论知识、算法/程序流程图、步骤和方法、关键源代码）**

**关键源代码：**

**// 计算最小支持度的事务数量**

**int totalTransactions = transactions.Count;**

**int minSupportCount = (int)(totalTransactions \* minSupport); // 支持度的最小数量**

**// 初始频繁项集**

**var frequentItemsets = transactions**

**.SelectMany(transaction => transaction)**

**.GroupBy(item => item)**

**.Where(group => group.Count() >= minSupportCount) // 只选择满足最小支持度的项集**

**.ToDictionary(group => new HashSet<string> { group.Key }, group => group.Count());**

**// 打印初始频繁项集**

**Console.WriteLine("初始频繁项集：");**

**foreach (var itemset in frequentItemsets)**

**{**

**Console.WriteLine($"{string.Join(", ", itemset.Key)}: {itemset.Value}");**

**}**

**int k = 2;**

**while (true)**

**{**

**// 生成 k-项集的候选项集**

**var candidateItemsets = GenerateCandidateItemsets(frequentItemsets.Keys, k);**

**// 如果没有候选项集，说明所有候选项集都已经生成完毕，退出循环**

**if (!candidateItemsets.Any())**

**{**

**Console.WriteLine("没有候选项集，终止.");**

**break;**

**}**

**// 计算候选项集的支持度**

**var candidateCounts = new Dictionary<HashSet<string>, int>();**

**foreach (var transaction in transactions)**

**{**

**var transactionSet = new HashSet<string>(transaction);**

**foreach (var candidate in candidateItemsets)**

**{**

**if (candidate.All(item => transactionSet.Contains(item))) // 如果项集在事务中出现**

**{**

**if (candidateCounts.ContainsKey(candidate))**

**candidateCounts[candidate]++;**

**else**

**candidateCounts[candidate] = 1;**

**}**

**}**

**}**

**// 只保留符合最小支持度的项集**

**var newFrequentItemsets = candidateCounts**

**.Where(kv => kv.Value >= minSupportCount)**

**.ToDictionary(kv => kv.Key, kv => kv.Value);**

**// 如果没有新的频繁项集，终止**

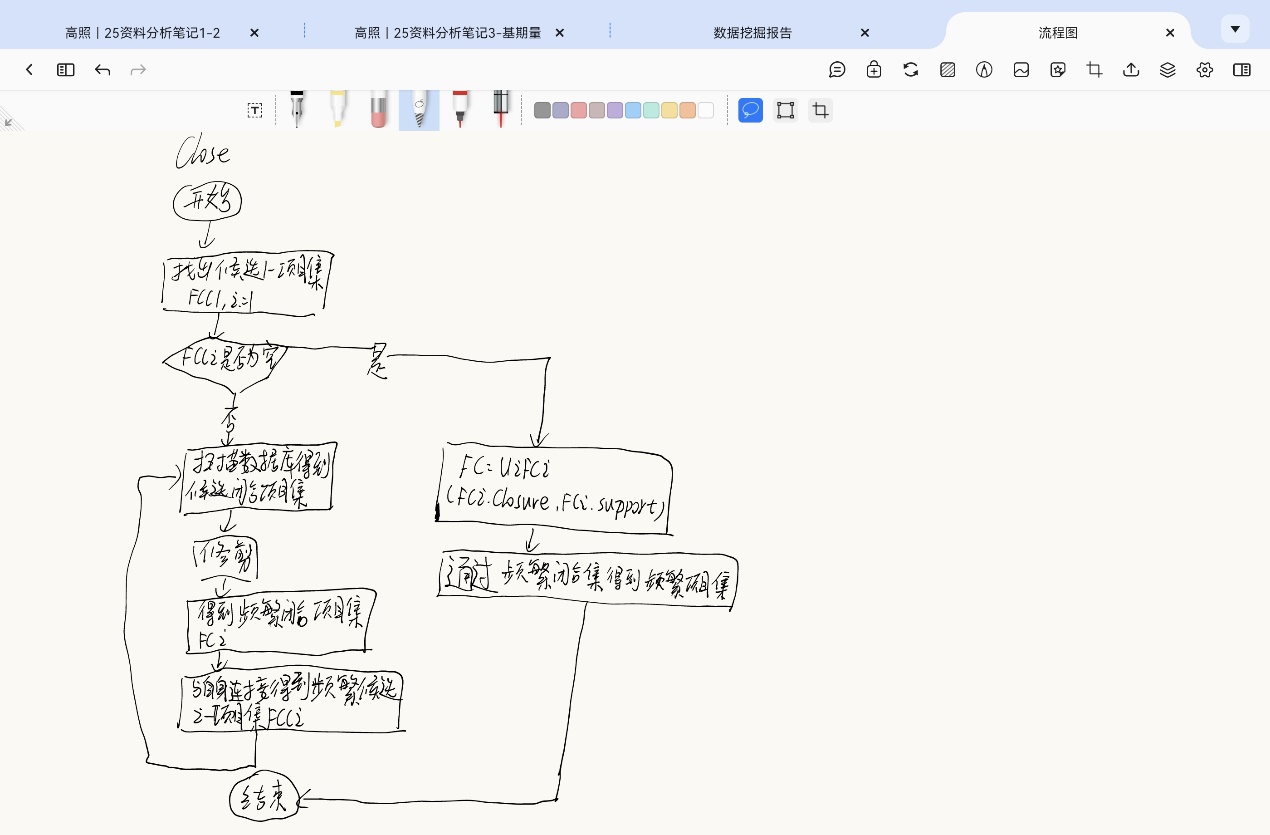
**if (!newFrequentItemsets.Any())**

**{**

**Console.WriteLine("没有找到新的频繁项集，终止.");**

**break;**

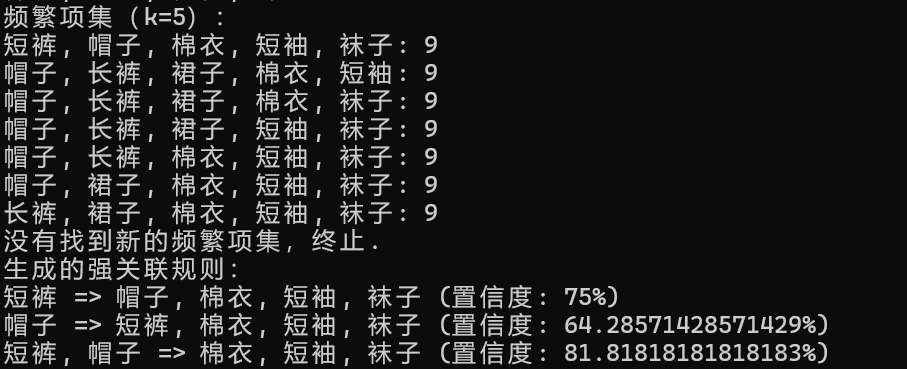
**}**

****

**Close算法流程图**

1. **实验结果与分析**

部分结果如下：



**五、小结与心得体会**

补充了Apriori算法的不足,学会了Close算法的原理和实现

1. **实验题目**

**实验三 FP-tree算法设计与应用**

1. **背景介绍**

Frequent Pattern Tree：不使用侯选集，直接压缩数据库成一个频繁模式树，通过频繁模式树可以直接得到频集。进行2次数据库扫描：一次对所有1-项目的频度排序；一次将数据库信息转变成紧缩内存结构。

1. **实验内容（包括：实验原理/运用的理论知识、算法/程序流程图、步骤和方法、关键源代码）**

**关键源代码： // 第一步：统计每个项的频率**

**var itemFrequency = transactions**

**.SelectMany(t => t)**

**.GroupBy(item => item)**

**.ToDictionary(g => g.Key, g => g.Count());**

**// 过滤掉不满足最小支持度的项**

**int minSupportCount = (int)Math.Ceiling(minSupport \* transactions.Count);**

**var filteredItems = itemFrequency**

**.Where(kv => kv.Value >= minSupportCount)**

**.OrderByDescending(kv => kv.Value)**

**.Select(kv => kv.Key)**

**.ToList();**

**// 第二步：构建FP-tree**

**FPTree fpTree = new FPTree();**

**foreach (var transaction in transactions)**

**{**

**var sortedTransaction = transaction**

**.Where(item => filteredItems.Contains(item))**

**.OrderBy(item => filteredItems.IndexOf(item))**

**.ToList();**

**fpTree.AddTransaction(sortedTransaction);**

**}**

**// 第三步：挖掘频繁项集**

**List<FrequentPattern> patterns = fpTree.MinePatterns(minSupportCount);**

**// 第四步：生成强关联规则**

**List<AssociationRule> rules = GenerateAssociationRules(patterns, minConfidence);**

**// 输出结果**

**Console.WriteLine("频繁项集:");**

**foreach (var pattern in patterns)**

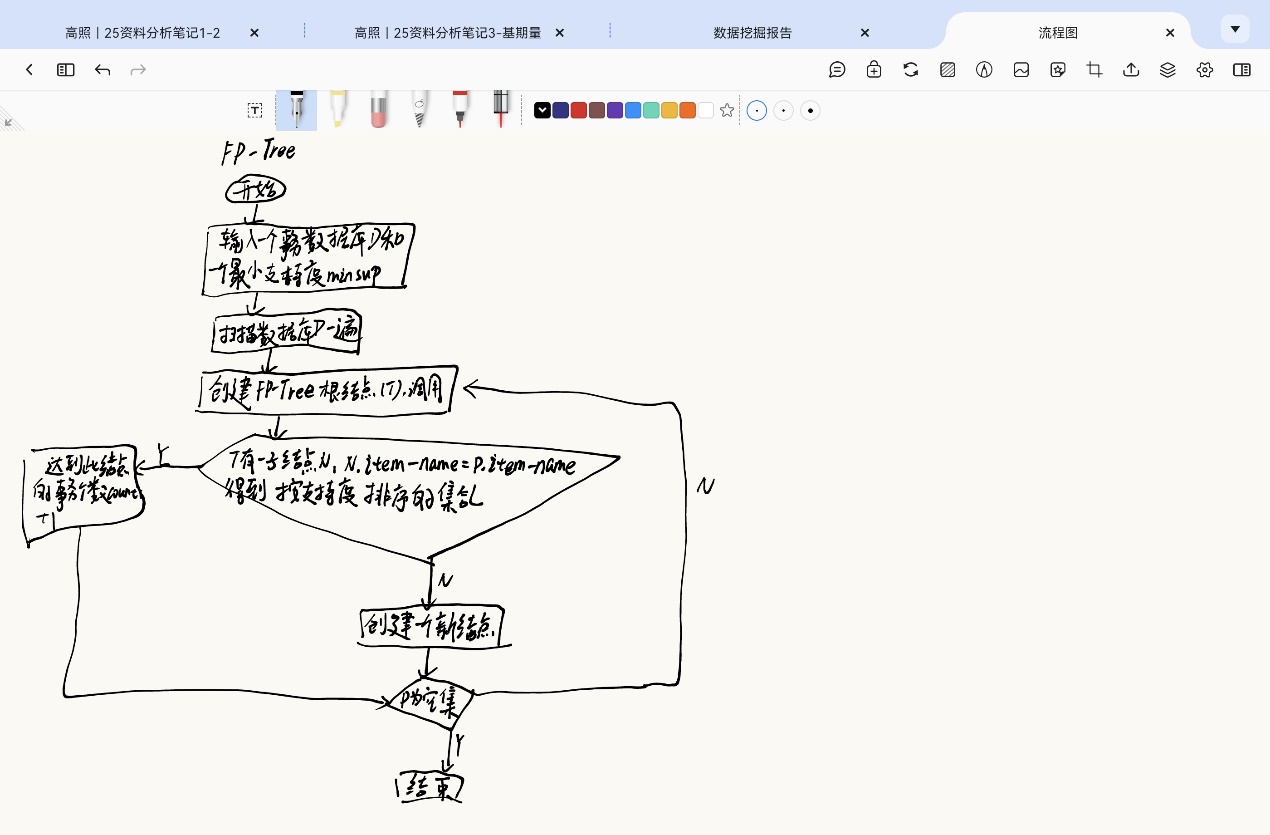
**Console.WriteLine($"{string.Join(",", pattern.Items)}: {pattern.Support}");**

**Console.WriteLine("\n强关联规则:");**

**foreach (var rule in rules)**

**Console.WriteLine($"{string.Join(",", rule.Antecedent)} => {string.Join(",", rule.Consequent)} (置信度: {rule.Confidence:P})");**

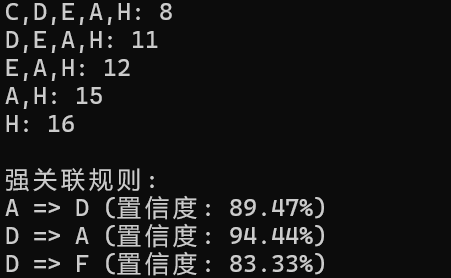
**}**

****

**FP-Tree流程图**

1. **实验结果与分析**

部分结果如下：



**五、小结与心得体会**

学会了FP-Tree的原理和实现

1. **实验题目**

**实验五KNN算法设计与应用**

1. **背景介绍**

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

1. **实验内容（包括：实验原理/运用的理论知识、算法/程序流程图、步骤和方法、关键源代码）**

**关键源代码：public static string PredictLevel(double height, List<Tuple<double, string>> data, int k)**

**{**

**// 1. 计算每个训练数据点与目标点之间的距离**

**var distances = new List<Tuple<double, string>>();**

**foreach (var record in data)**

**{**

**double distance = Math.Abs(record.Item1 - height); // 计算欧几里得距离**

**distances.Add(Tuple.Create(distance, record.Item2));**

**}**

**// 2. 按距离升序排序**

**distances.Sort((x, y) => x.Item1.CompareTo(y.Item1));**

**// 3. 选择最近的k个邻居**

**var nearestNeighbors = distances.Take(k).ToList();**

**// 4. 投票选择最多的类别**

**var voteCounts = nearestNeighbors.GroupBy(x => x.Item2)**

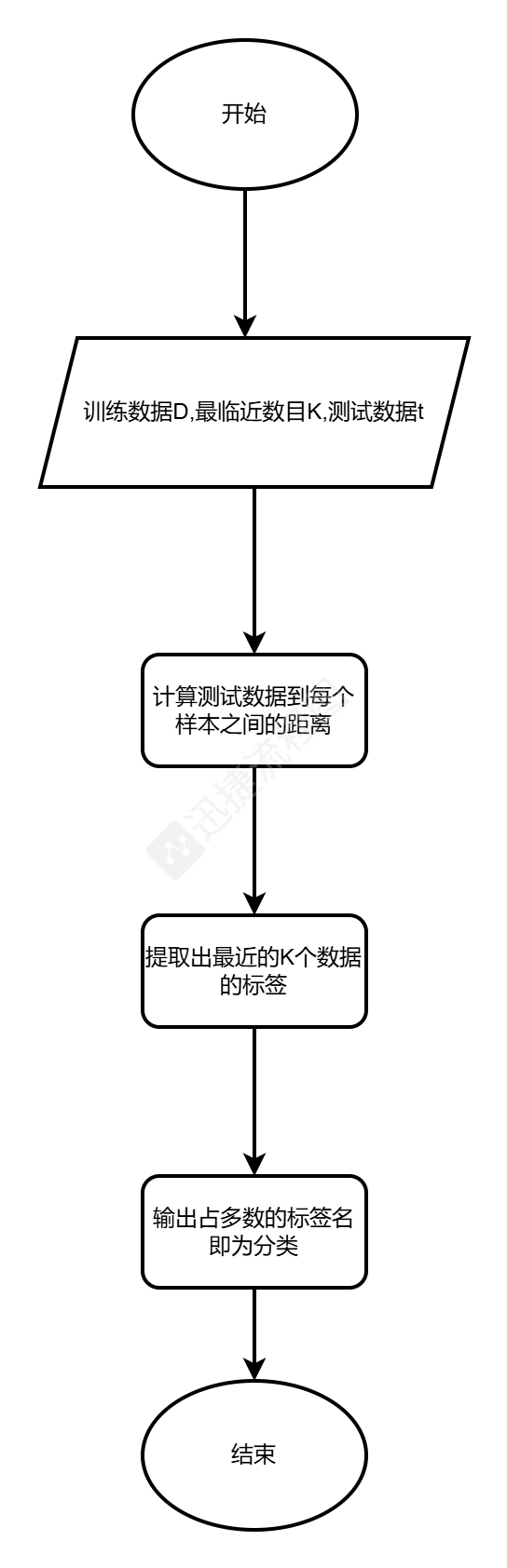
**.ToDictionary(g => g.Key, g => g.Count());**

**// 5. 返回出现最多的类别**

**var predictedClass = voteCounts.Aggregate((x, y) => x.Value > y.Value ? x : y).Key;**

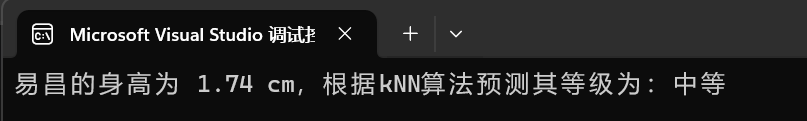
**return predictedClass;**

**}**

****

**KNN流程图**

1. **实验结果与分析**

输出结果如下：

**五、小结与心得体会**

学会了KNN算法的原理和实现

1. **实验题目**

**实验六 ID3算法设计与实现**

1. **背景介绍**

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

1. **实验内容（包括：实验原理/运用的理论知识、算法/程序流程图、步骤和方法、关键源代码）**

关键源代码如下：

// 构建决策树

public static TreeNode BuildDecisionTree(List<DataSet> data, List<string> attributes)

{

// 如果数据集为空或所有实例的标签相同，返回叶节点

if (data.Count == 0)

return null;

var firstLabel = data[0].BuyComputer;

if (data.All(d => d.BuyComputer == firstLabel))

{

return new TreeNode { Label = firstLabel };

}

// 如果没有特征可供选择，返回叶节点

if (attributes.Count == 0)

{

return new TreeNode { Label = firstLabel };

}

// 选择最佳属性

string bestAttribute = ChooseBestAttribute(data, attributes);

var node = new TreeNode { Attribute = bestAttribute };

// 递归创建子树

var subsets = SplitDataSet(data, bestAttribute);

foreach (var subset in subsets)

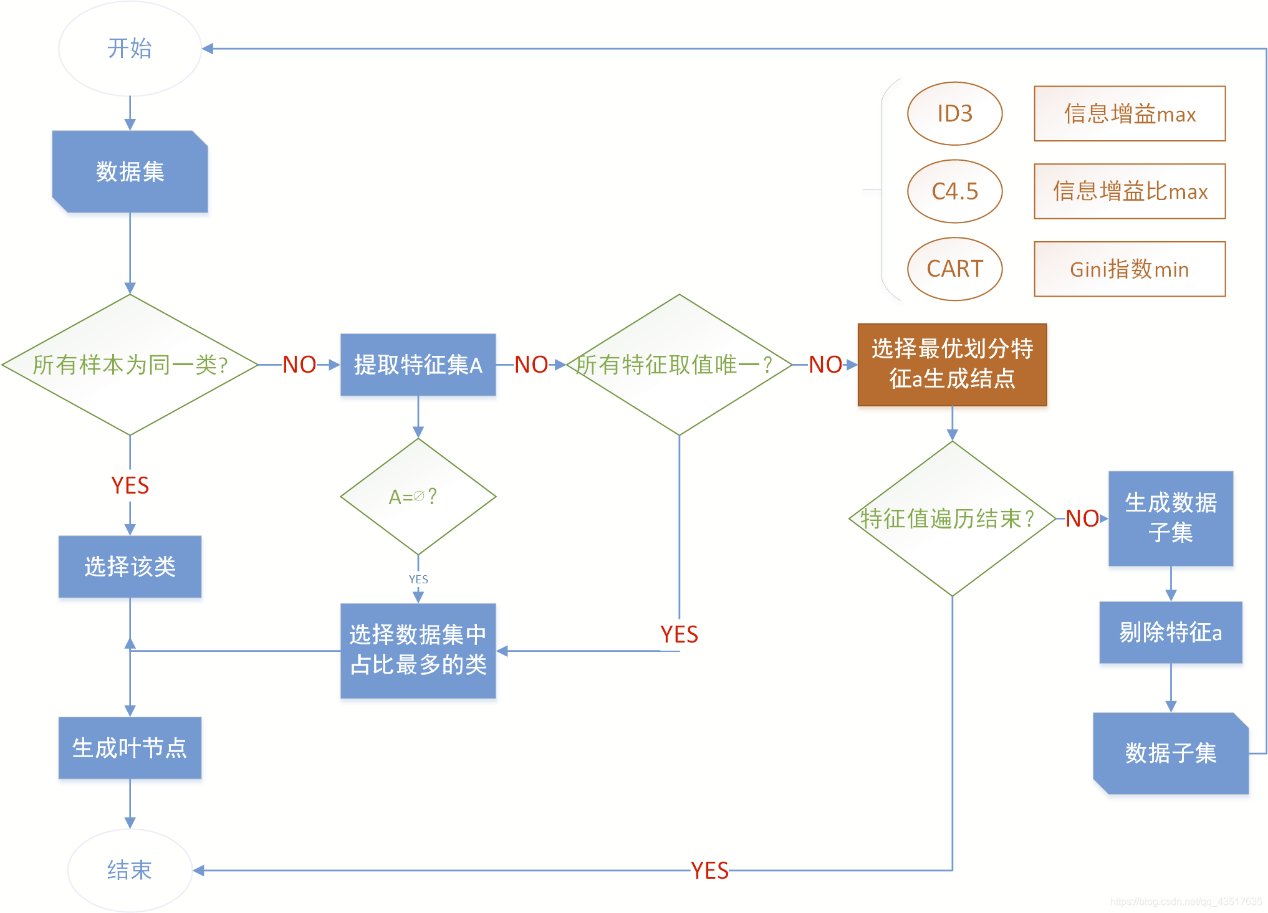
{

var remainingAttributes = new List<string>(attributes);

remainingAttributes.Remove(bestAttribute);

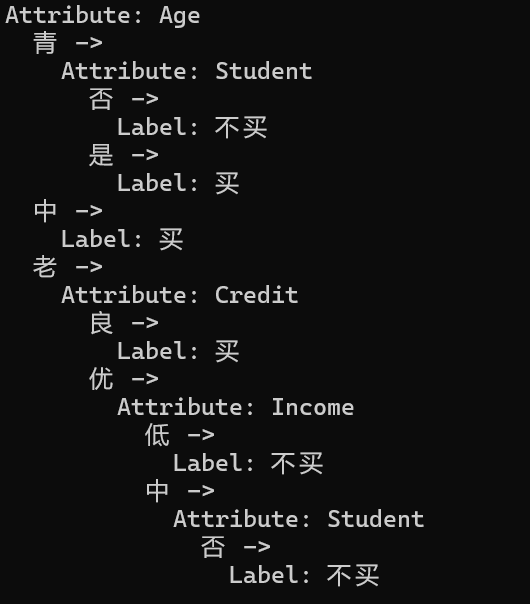
node.Children[subset.Key] = BuildDecisionTree(subset.Value, remainingAttributes);

return node;

}

ID3流程图

1. **实验结果与分析**

最终结果输出如下: 

**五、小结与心得体会**

学会ID3算法原理和实现