# 图书馆借阅信息数据仓库的设计与构建

**林晋霆 网络工程 2014051795**

目录

[图书馆借阅信息数据仓库的设计与构建 1](#_Toc485557654)

[1. 简介 1](#_Toc485557655)

[2. 数据仓库需求分析与模型设计 2](#_Toc485557656)

[2.1 需求分析 2](#_Toc485557657)

[2.2 概念模型设计 2](#_Toc485557658)

[2.3 逻辑模型设计 4](#_Toc485557659)

[2.4 物理模型设计 5](#_Toc485557660)

[3. 原始数据的 ETL 与数据仓库的实现 5](#_Toc485557661)

[3.1 数据挖掘算法的选择 5](#_Toc485557662)

[3.2 原始数据的 ETL 6](#_Toc485557663)

[3.3 数据挖掘系统的实现 7](#_Toc485557664)

[4. 挖掘结果的展示与未来应用的探讨 9](#_Toc485557665)

## 简介

随着图书馆借阅数据的不断积累，使用人工方式对借阅数据进行管理已经无法满足实际需求。在结合发达的计算机技术后，通过数据仓库对积累的海量图书借阅数据进行检索与分析已经成为了当前管理海量数据的唯一方式。

通过对图书馆海量借阅数据的挖掘，我们可以获取隐藏在其中的信息，并分析结果来对未来的决策进行支持。如通过 FP-Growth 算法来发现书籍之间的频繁项，进而在用户检索可借阅书籍的时候进行针对性的推荐，以提高图书的流通率

## 数据仓库需求分析与模型设计

在建立一个数据仓库之前，我们首先要完成需求分析以及数据仓库的概念模型、逻辑模型和物理模型的设计。

### 需求分析

显然，构建此数据仓库的目的在于管理和分析已有的海量借阅数据，并在用户发起查询请求的时候即时提供计算结果。那么，从这个起点出发，我们就可以得到以下需求点：

1. 通过分析提供的原始数据来计算其中包含的频繁项信息；
2. 需要对提供的原始数据进行高效的ETL操作；
3. 在调用者发出查询请求的1秒以内完成与本次查询有关的分析计算工作，并以适当的形式把计算结果返回到调用者一方；
4. 提供信息全面、形式得当的报表供管理员确认系统的工作状态；
5. 其他可能的合理需求。

### 概念模型设计

本节使用E-R图对所获得的本数据仓库进行建模。



**图2-1** 借阅记录数据仓库中的概念模型

图2-1所示的概念模型所代表的操作为：

1. 调用者（管理员或应用服务器）向OLAP服务器提交包含期望结果类别的查询要求；
2. OLAP服务器根据请求中的类别从数据仓库中抽取对应类别的数据，并计算频繁项集；
3. OLAP服务器把计算结果以约定的形式返回到调用者一方。

因此，我们可以从此概念模型中抽象出“用户查询请求”、“计算结果”和“原始数据”三个主题。在本数据仓库的预想使用场景中，“用户查询请求”来自于外部数据，故不需要在数据仓库中进行装载；“计算结果”来自于对原始数据的计算，也不需要在数据仓库中专门划分出空间储存；故本数据仓库需要装载的主题为“原始数据”。基于原始数据，我们就能很方便地根据调用者的请求而计算出所需结果。

在“原始数据”这一主题中，每一项均包含了以下属性：



**图2-2** 数据项E-R图

在原始数据中，每一册图书均有不重复的“图书记录号（册）”属性，而每一种（即所有ISBN相同的书籍）均有不重复的“图书记录号（种）”属性。故我们选定“图书记录号（册）”作为原始数据的主键。同时把主键和“图书分类号”添加到索引中，以加快查找速度。

### 逻辑模型设计

2.2节已经分析出本数据仓库需要装载的主题为“原始数据”，那么我们从所获的样例数据集出发，那么很自然地，我们就可以得到以下三种对数据进行分类计算的粒度以及在此粒度下的数据预计使用场景：

**表2-1** 数据粒度分类详细信息表

|  |  |  |
| --- | --- | --- |
| **粒度** | **用途** | **大小** |
| 大粒度 | 以年为单位的报告性分析所用 | 每单位约27万 |
| 中粒度 | 以月为单位的即时计算所用 | 每单位约2万 |
| 小粒度 | 以项为单位的详细报告所用 | 每单位为1项 |

在数据分割方面，我们采用与表2-1相同的粒度对数据进行分割。

在底层记录模式方面，本数据仓库不对数据记录增加导出记录；同时，底层数据存储模式采用与原始数据相同的关系型表模式。

### 物理模型设计

本节主要涉及本数据仓库的索引策略和数据存储策略。

常用的索引有B-Tree索引和位图索引两种。在本数据仓库中，要索引的属性有“图书分类号（种）”和“图书记录号”两个。由于这两个属性的取值十分多样化，故取值区间较为单一的位图索引不适用于此。同时，这两个属性的值在全数据库中，每一类书都有独特的取值，故本数据仓库采用B-Tree索引。

在数据存储策略上，本数据仓库所涉及的数据量较大。同时，原始数据中有许多字段在数据挖掘过程（通过PF-Growth计算频繁项）中是不需要使用的。故使用分割表策略。即从原始数据中抽取出“计算输入”表，其中存放ETL过程所需要的“读者记录号”、“图书记录号（种）”以及“图书分类号”三项。其余的数据存放在“原始数据”表中，以加快ETL过程。

## 原始数据的 ETL 与数据仓库的实现

本节主要讨论原始数据的ETL过程（即原始数据的提取、转换、清洗和加载四个过程）以及数据仓库的具体实现方式。

### 数据挖掘算法的选择

由于本数据仓库的主要输出结果为输入数据中的频繁项信息，而常用的频繁项挖掘算法有Apriori算法和FP-Growth算法。但我们所需要计算的数据量过大，与FP-Growth算法相比，Apriori算法又需要对内存或者数据库进行多次扫描才能得出结果，算法时空复杂度较大，故不予采用。虽然FP-Growth算法在实现上较为麻烦，但考虑到数据量的问题，这种编写时候的付出还是值得的。

### 原始数据的 ETL

2.2节和2.4节已经讨论了原始数据的属性集以及数据的抽取。那么，要实现原始数据到计算输入这一ETL过程，就需要把原始数据“重整”为FP-Growth算法所能识别的数据输入。

* + 1. **数据的提取与转换**

在数据的提取上，由于本数据仓库的原始数据存放在数据库中，我们可以直接从“原始数据”表中提取“读者记录号”、“图书记录号（种）”以及“图书分类号”三列到“计算输入”表中，然后进行ETL过程的其余步骤。同时，由于数据库能直接以查询结果集的形式提供原始数据，那么也不需要对原始数据进行格式转换。

* + 1. **数据的清洗与加载**

由于FP-Growth算法的输入为形如“Item1 Item2 Item3”这样以行为单位的一个结果集，那么我们就需要把数据重整为这样的一种格式。具体的算法为：

**for** (i = 0; i < inputItemCount; i++) {

merged.add(result.get(i).get(1));

**for** (j = i; j < inputItemCount - 1; j++) {

**if** (Objects.equals(result.get(j).get(0), result.get(j + 1).get(0))) {

merged.add(result.get(j + 1).get(1));

} **else** {

**break**;

}

}

i = j;

**if** (merged.size() == 1) {

merged.clear();

**continue**;

}

Collections.sort(merged);

FPInput.add(**new** ArrayList<>(merged));

merged.clear();

}

以上算法把原始数据（即result）从上之下扫描。在扫描的过程中，如果遇到“读者记录号”相同的几项，就会把他们合并到同一行中，即算法中的merged；如果合并后的merged大小为1，则认为这一行只有一个结果，便舍弃之（FP-Growth算法要求有意义的输入数据每一行至少有两条记录）。然后把FPInput直接输入到算法中即可开始计算过程。

### 数据挖掘系统的实现

在实现上，我们把这个数据仓库对原始数据的处理大致分成三个阶段：

1. **预处理阶段**

fpInput = preProcess(csvFilePath, inputSeparator, requiredClass);

算法输入：

1. String csvFilePath：以 CSV 文件格式存储的原始数据

2. String inputSeparator：CSV 文件中用来分隔同一行中的不同数据项的分隔符

3. String targetClass：目的图书分类

算法输出：

List<List<String>> FPInput：FP-Growth 算法输入数据

List<List<String>> preProcess(String csvFilePath, String inputSeparator, char targetClass) {

//1.读取 CSV 文件，构建原始输入 ...

//2.筛选出符合类别要求的记录项 ...

//3.读取数据并合并同一次结束记录的连续几行记录 ...

return FPInput;

}

本阶段主要实现对原始数据的ETL过程（3.2节）。即对原始数据进行抽取、清洗以及加载，使之成为符合FP-Growth算法要求的输入。

1. **频繁项计算阶段**

FPGrowth fpGrowth = new FPGrowth(2);  
fpGrowth.getFPOutput(fpInput, null, fpOutput);

本阶段主要使用FP-Growth算法对输入数据中的频繁项的计算。

1. **重整输出阶段**

result = reProcess(dictionaryFilePath, fpSeparator, fpOutput)

算法输入：

1. String dictionaryFilePath：CSV 文件格式的书名字典文件的位置

2. String dictionarySeparator：字典文件中每一行的各项之间的分隔符

3. List<List<String>> fpOutput：欲处理的 FP-Growth 输出

算法输出：

List<List<String>> result：可以直接输出的结果

List<List<String>> reProcess(String dictionaryFilePath, String dictionarySeparator, List<List<String>> fpOutput) {

//1.制作字典

Dictionary<String, String> dictionary = new Hashtable<>();

...

//2.合并与去重

getResultFiltered(result, fpOutput);

//3.替换其中的书号以得到最终结果

...

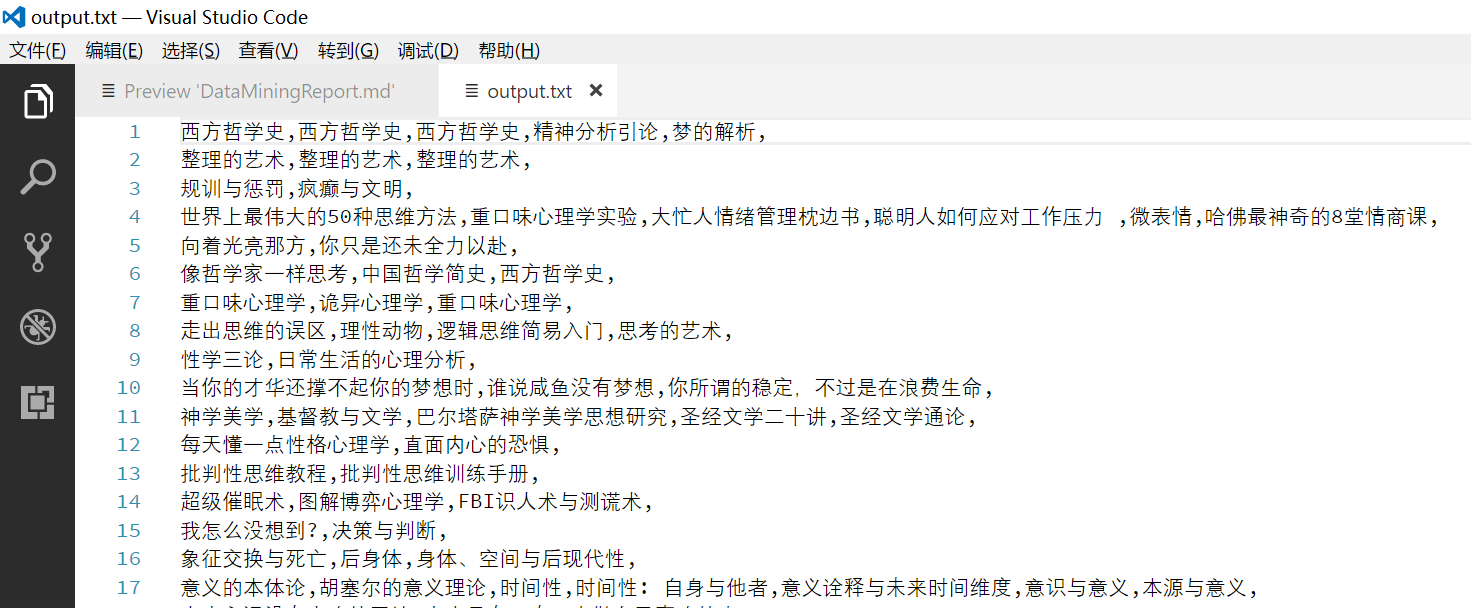
return result;

}

本阶段把FP-Growth算法的单项结果尽可能地扩大。如频繁项集A含“1，2，3”共三项，B含“1，2，3，4”共四项。那么，要使得结果集最大化，为用户提供最多样的结果，就需要把A和B两项合并。同时，本阶段还需要把结果集中的“图书记录号（种）”替换为相应的书名，以供显示之用；同时还需要保留原始的“图书记录号（种）”，以供程序查询之用。

## 挖掘结果的展示与未来应用的探讨

获得的结果如图4-1所示：、



**图4-1** 实验结果

图4-1中的结果都是根据B类书籍的借阅记录挖掘出的书名频繁项。通过这些频繁项以及对应的“图书记录号（种）”信息，我们可以在搜索结果也中根据用户所查询书籍的类别，给予对应类别的推荐结果。

# FP-Growth 和 K-Means 实验报告

## 1. FP-Growth

在生活中，我们常常会遇到一些需要分析事物之间的关联性的场合。例如，在分析超市的销售数据时，我们可能会想知道，顾客在买牛奶的时候，还会买什么别的东西。还有数据挖掘领域里面著名的啤酒与尿布的故事R1。

要解决这些问题，我们就需要一种算法来帮我们寻找这些事务项之间的关联性。常用的**关联分析（Association Analysis）** 算法有 Apriori 算法和 FP-Growth 算法。Apriori 算法的时空复杂度都比较高，现在已经不常用了，故本文略去对 Apriori 算法的介绍，专注于对 FP-Growth 的介绍与分析。

### 1.1 支持度计数筛选

在 FP-Growth 算法里面，需要对每一个事项计算各自的支持度计数（即此事务在全集中出现的次数）。如果支持度不满足设定的最小值，那么这项记录将不能被算法所收录。

### 1.2 步骤简介

FP-Growth 的步骤相对于 Apriori 会简单一点，但绝对值也不低。其伪代码步骤简介如下[5]：

输入：事务集合 List<List<String>> transactions

输出：频繁模式集合 List<List<String>> fpOutput

public void getFPOutput(List<List<String>> transactions, List<String> postPattern, List<List<String>> fpOutput) {

构建头表项 HeaderTable：buildHeaderTable(transactions);

构建 FP 树：buildFPTree(headerTable, transactions);

if (树空) return;

输出频繁项集;

遍历每一个头项表节点并递归;

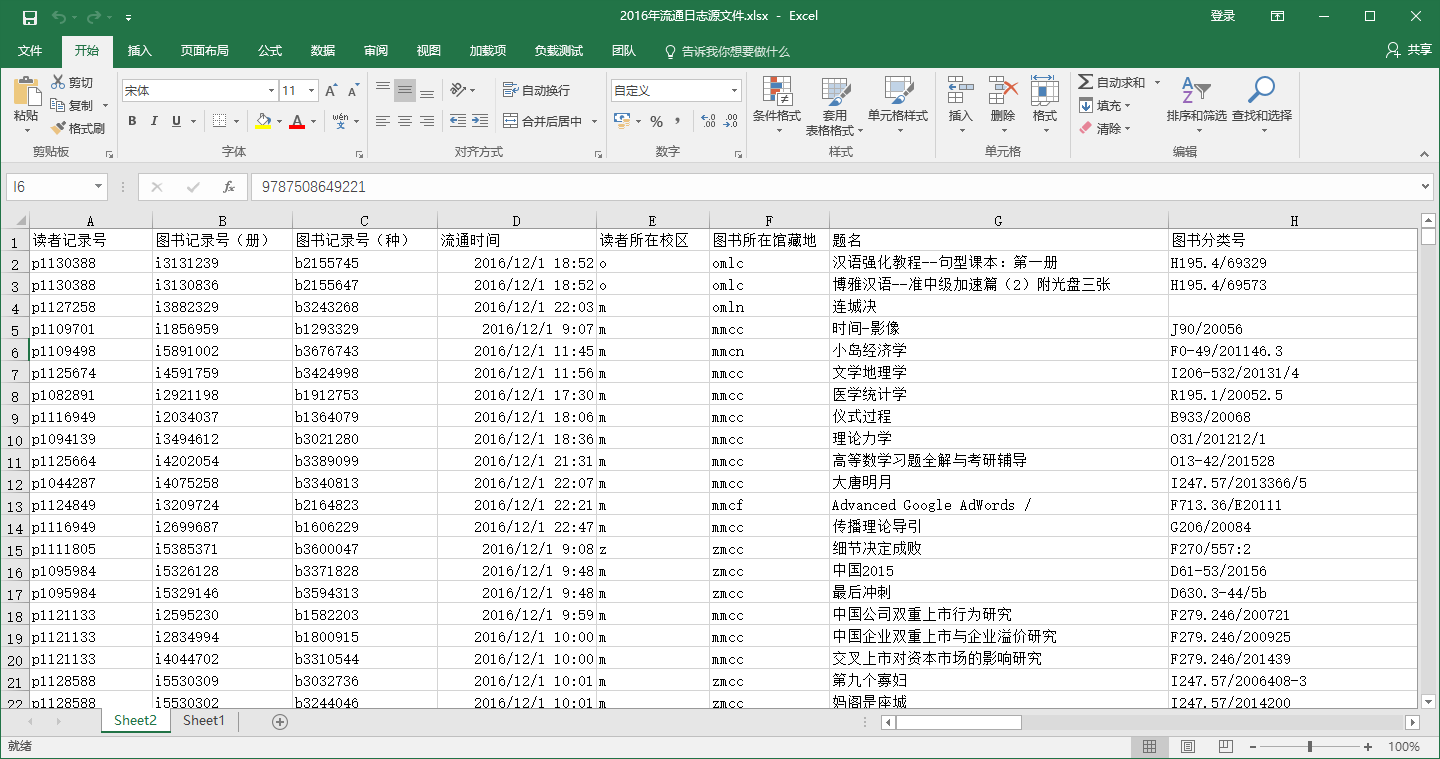
}

具体操作步骤请参考源代码。

### 1.3 实例分析

#### 1.3.1 Overview

针对 FP-Growth 的实例分析，我们采用了一个具有 27 万测试数据的数据集（示例见 Fig.1，可以通过[度盘链接](http://pan.baidu.com/s/1eRZ0vke)下载）。在经过预处理阶段之后（即源代码中的 preProcess 方法），数据量下降为 6 万多，全过程处理时间大约为 10 秒。不同机器可能需要不同的处理时间，具体请参照在程序起止是输出的时间戳。



**Fig.1** Data Sample

测试程序主要采用了三级处理的方式，预处理、FP-Growth 计算频繁项集和重整输出三个阶段：

1. 预处理：  
   fpInput = preProcess(csvFilePath, inputSeparator, requiredClass);
2. FP-Growth 算法生成频繁项：  
   FPGrowth fpGrowth = new FPGrowth(2);  
   fpGrowth.getFPOutput(fpInput, null, fpOutput);
3. 重整输出：  
   result = reProcess(dictionaryFilePath, fpSeparator, fpOutput)

由于频繁项的计算已在 1.2 节中介绍，故以下只介绍预处理和重整输出两个阶段。

##### 1.3.1.1 预处理

预处理阶段的主要任务是为 FP-Growth 准备需要的输入数据。

如前文所述，原始数据为日志文件，其数据项按行排列，并非 FP-Growth 所要求的多个事物项处在同一行。故预处理阶段的第一个任务就是把这些“属于”同一行的数据全部合并到同一行中：

算法输入：

1. String csvFilePath：以 CSV 文件格式存储的原始数据

2. String inputSeparator：CSV 文件中用来分隔同一行中的不同数据项的分隔符

3. String targetClass：目的图书分类

算法输出：

List<List<String>> FPInput：FP-Growth 算法输入数据

List<List<String>> preProcess(String csvFilePath, String inputSeparator, char targetClass) {

//1.读取 CSV 文件，构建原始输入

...

//2.筛选出符合类别要求的记录项

...

//3.读取数据并合并同一次结束记录的连续几行记录

...

return FPInput;

}

##### 1.3.1.2 重整输出

重整输出阶段的任务就是把 FP-Growth 算法输出的杂乱无章的结果重整为在条目之间具有唯一性的输出结果。

算法输入：

1. String dictionaryFilePath：CSV 文件格式的书名字典文件的位置

2. String dictionarySeparator：字典文件中每一行的各项之间的分隔符

3. List<List<String>> fpOutput：欲处理的 FP-Growth 输出

算法输出：

List<List<String>> result：可以直接输出的结果

List<List<String>> reProcess(String dictionaryFilePath, String dictionarySeparator, List<List<String>> fpOutput) {

//1.制作字典

Dictionary<String, String> dictionary = new Hashtable<>();

...

//2.合并与去重

getResultFiltered(result, fpOutput);

//3.替换其中的书号以得到最终结果

...

return result;

}

其中，getResultFiltered(List<List<String>> result, List<List<String>> removedPrefixs) 会在下一节详细介绍。

#### 1.3.2 如何把 FP-Growth 算法的输出还原成可阅读的频繁项集？

FP-Growth 算法输出是杂乱无章的，所以我们就需要对它的输出进行重整。而为了尽可能扩大相关联事务的范围，我们采用了合并所有有交集的行的方法：

算法输入：

1. List<List<String>> result：处理结果

2. List<List<String>> fpOutput：FP-Growth 处理结果与应用场合

算法输出：

在参数 result 中

private static void getResultFiltered(List<List<String>> result, List<List<String>> fpOutput) {

//合并与去重

HashSet<String> temp = new HashSet<>();

int elementLeft = fpOutput.size();

//任意两行之间如果有交集，就合并他们

while (fpOutput.size() != 0) {

temp.addAll(fpOutput.get(0));

for (int j = 1; j < elementLeft - 1; j++) {

//若两项之间有交集，则 isIntersected 返回 true

if (isIntersected(temp, fpOutput.get(j))) {

temp.addAll(fpOutput.get(j));

fpOutput.remove(j);

elementLeft--;

j--;

}

}

result.add(new ArrayList<>(temp));

temp.clear();

elementLeft--;

fpOutput.remove(0);

}

}

以上算法中，HashSet 的使用有效加快了匹配的速度。同时，由于算法会删去已添加的行，重整算法的时间复杂度近似为 O(nlogn)，空间复杂度不会超过输入数组的大小，即 O(n)。

#### 1.3.3 如何判断两个频繁项之间有交集？

为了能尽可能地扩大结果中一条频繁项集的数据，为推荐算法提供更多样化的结果，在合并频繁项集的时候，需要判断两个频繁项之间是否有交集。如果两个频繁项之间有交集，则合并两者。

算法输入：

1. HashSet<String> a：已有结果的条件模式基

2. List<String> b：需要检测的频繁项集

算法输出：

true：两者之间有交集

false：两者之间无交集

private static boolean isIntersected(HashSet<String> a, List<String> b) {

for (String anA : a) {

for (String aB : b) {

if (anA.equals(aB)) {

return true;

}

}

}

return false;

}

据其他资料分析，使用 List 的 retainAll() 方法也能检测两者之间是否有交集，大家可以去试试这种方法。

#### 1.3.4 处理结果与应用场合

本算法能在给定的事务集中高效计算频繁项集。那么，我们就能把这个算法移植到服务器端，并在小数据量的情况下实现根据用户的指定的类别，实时计算频繁项集，并在结果页面推荐给用户。

## 2. K-Means

在对数据进行了关联分析之后，有时候还需要对数据进行**聚簇分析（Clustering Analysis）**。聚类分析的算法较多，这里只介绍 K-Means 算法。这个算法的输入有数据集和分类数目 K；输出是分在 K 个簇中的数据项。

### 2.1 分类原理

分类主要涉及计算欧几里得距离和计算一群质点的质心两个算法，下面分别介绍：

#### 2.1.1 欧几里得距离的高效计算

分类的方法主要是计算某个点与所有 K 个质心之间的欧几里得距离。计算两个 n 维点之间的欧几里得距离[4]：

C:\Users\K9A2S\AppData\Local\Microsoft\Windows\INetCache\Content.Word\n维欧几里得距离.jpg

**Fig.2** EuclidDistance

实用的计算方法：

private static double getEuclidDistance(Point a, Point b) {

double result = 0;

result += Math.pow(a.getX() - b.getX(), 2);

result += Math.pow(a.getY() - b.getY(), 2);

result += Math.pow(a.getZ() - b.getZ(), 2);

return Math.sqrt(result);

}

#### 2.1.2 计算质心的方法：

在 K-Means 算法中，分类需要按照欧几里得距离最小的原则。但在实用的算法中，通常采用重心来代替质心：

private static Point getClusterCenter(List<Point> points) {

if (points.size() == 0) {

return null;

}

if (points.size() == 1) {

return new Point(points.get(0).getX(), points.get(0).getY(), points.get(0).getZ());

}

double x = 0;

double y = 0;

double z = 0;

for (Point point : points) {

x += point.getX();

y += point.getY();

z += point.getZ();

}

x = x / points.size();

y = y / points.size();

z = z / points.size();

return new Point(x, y, z);

}

### 2.2 步骤简介

K-Means 算法是一种很好理解的算法，其步骤异常简单。

1. 用户提供输入数据集。数据集中的每一项都需要包含若干属性。如输入一个二维点集，那其中的一项就需要至少包含 X 和 Y 两个坐标；
2. 由用户指定初始质心或者由算法在输入的数据集中随机选取 K 个点作为初始质心；
3. 计算每一项到每一个质心之间的欧几里得距离；
4. 按照欧几里得距离最小的原则，把这些点分到 K 个簇中的某一个；
5. 重新计算 K 个簇中的质心（通常用计算重心代替）；
6. 如果质心与分类时使用的质心相同，则算法结束；否则就需要重复 2-6 步。

### 2.3 计算实例

由于每一个点不仅仅需要保存自身的三轴坐标，同时还要保存自身的类别以及名字，故新建了用于表示点的 Point 类，并以 Point[] 来表示点集。

计算实例采用了 CPDA 数据分析天地提供的足球数据[3]。由于设计的时候采用了三维点集，所以无法采用通常的二维分类着色图R2来表示，故直接输出三种分类。其实验结果如下：

*日本,韩国,澳大利亚,*

*印尼,泰国,*

*中国,朝鲜,伊拉克,伊朗,沙特,阿联酋,卡塔尔,乌兹别克斯坦,巴林,阿曼,约旦,*

其结果符合球队实际排位。

另外，由于本次实验中尚未添加对分类的排序功能，即在输出的时候并非按照质心的“权重”来进行排序，故输出的结果是不能直接提取到别的程序中的。

### 2.4 实现要点

#### 2.4.1 随机初始质心的获取

获取随机初始质心有两种方法：

1. 第一种是采用 Collection.shuffle() 来直接打乱排序，然后直接取前 K 位作为随机初始质心；
2. 第二种就是使用随机数。先计算出 K 个不重复的随机数，然后按照获得的随机数到 Point[] 数组中获取随机初始质心，其计算过程如下：

private static int[] getUnrepeatedRandomNumbers(int min, int max, int count) {

int[] result = new int[count];

int i = 0;

HashSet<Integer> temp = new HashSet<>();

while (true) {

if (temp.size() == count) {

break;

}

temp.add((int) (Math.random() \* (max - min)) + min);

}

for (Integer item : temp) {

result[i] = item;

i++;

}

return result;

}

#### 2.4.2 Point[] 数组的复制与判同

1. 如果直接调用系统提供的 System.arraycopy(b, 0, a, 0, a.length)，那在复制的时候就是浅复制：即两个数组都是“引用了”同一个来源。在其中一个数组被改变的时候，另外一个数组由于引用了同一块内存区域，其值也会被改变。故要实现数组的“深拷贝”，则需要自行编写复制方法：

private static Point[] getArrayCopy(Point[] b) {

Point[] a = new Point[b.length];

if (a.length == 0 || b.length == 0) {

return null;

}

System.arraycopy(b, 0, a, 0, a.length);

return a;

}

1. 那么如何判断两个质心是否移动呢？我们可以直接采用逐行判断的方式：

private static boolean isClusterCenterChanged(Point[] a, Point[] b) {

for (int i = 0; i < a.length; i++) {

if (a[i].getX() != b[i].getX()) {

return true;

} else if (a[i].getY() != b[i].getY()) {

return true;

} else if (a[i].getZ() != b[i].getZ()) {

return true;

}

}

return false;

}

在任意一次比较中，如果两者的 X、Y 和 Z 三个值中的任意一个不相等，方法就会返回 true，即质心已移动；否则返回 false，表示质心未移动。

## 3.Reference

1. [Grant Stanley - Diapers, Beer, and Data Science in Retail](http://canworksmart.com/diapers-beer-retail-predictive-analytics/)
2. [听云博客 - JAVA实现K-means聚类](http://www.tuicool.com/articles/VBBnie)
3. [CPDA 数据分析天地 - 用K-means看透中国男足！](http://www.sohu.com/a/135368994_354986)
4. [tianlan\_new\_start - 欧几里得距离、曼哈顿距离和切比雪夫距离](http://blog.csdn.net/tianlan_sharon/article/details/50904641)
5. [人非木石\_xst - 单机和集群环境下的FP-Growth算法java实现(关联规则挖掘)](http://blog.csdn.net/shimin520shimin/article/details/49281381)