# 探索推荐引擎内部的秘密，第 3 部分: 深入推荐引擎相关算法 - 聚类

## 聚类分析

### 什么是聚类分析？

聚类 (Clustering) 就是将数据对象分组成为多个类或者簇 (Cluster)，它的目标是：在同一个簇中的对象之间具有较高的相似度，而不同簇中的对象差别较大。所以，在很多应用中，一个簇中的数据对象可以被作为一个整体来对待，从而减少计算量或者提高计算质量。

其实聚类是一个人们日常生活的常见行为，即所谓“物以类聚，人以群分”，核心的思想也就是聚类。人们总是不断地改进下意识中的聚类模式来学习如何区分各个事物和人。同时，聚类分析已经广泛的应用在许多应用中，包括模式识别，数据分析，图像处理以及市场研究。通过聚类，人们能意识到密集和稀疏的区域，发现全局的分布模式，以及数据属性之间的有趣的相互关系。

聚类同时也在 Web 应用中起到越来越重要的作用。最被广泛使用的既是对 Web 上的文档进行分类，组织信息的发布，给用户一个有效分类的内容浏览系统（门户网站），同时可以加入时间因素，进而发现各个类内容的信息发展，最近被大家关注的主题和话题，或者分析一段时间内人们对什么样的内容比较感兴趣，这些有趣的应用都得建立在聚类的基础之上。作为一个数据挖掘的功能，聚类分析能作为独立的工具来获得数据分布的情况，观察每个簇的特点，集中对特定的某些簇做进一步的分析，此外，聚类分析还可以作为其他算法的预处理步骤，简化计算量，提高分析效率，这也是我们在这里介绍聚类分析的目的。

### 不同的聚类问题

对于一个聚类问题，要挑选最适合最高效的算法必须对要解决的聚类问题本身进行剖析，下面我们就从几个侧面分析一下聚类问题的需求。

**聚类结果是排他的还是可重叠的**

为了很好理解这个问题，我们以一个例子进行分析，假设你的聚类问题需要得到二个簇：“喜欢詹姆斯卡梅隆电影的用户”和“不喜欢詹姆斯卡梅隆的用户”，这其实是一个排他的聚类问题，对于一个用户，他要么属于“喜欢”的簇，要么属于不喜欢的簇。但如果你的聚类问题是“喜欢詹姆斯卡梅隆电影的用户”和“喜欢里奥纳多电影的用户”，那么这个聚类问题就是一个可重叠的问题，一个用户他可以既喜欢詹姆斯卡梅隆又喜欢里奥纳多。

所以这个问题的核心是，对于一个元素，他是否可以属于聚类结果中的多个簇中，如果是，则是一个可重叠的聚类问题，如果否，那么是一个排他的聚类问题。

**基于层次还是基于划分**

其实大部分人想到的聚类问题都是“划分”问题，就是拿到一组对象，按照一定的原则将它们分成不同的组，这是典型的划分聚类问题。但除了基于划分的聚类，还有一种在日常生活中也很常见的类型，就是基于层次的聚类问题，它的聚类结果是将这些对象分等级，在顶层将对象进行大致的分组，随后每一组再被进一步的细分，也许所有路径最终都要到达一个单独实例，这是一种“自顶向下”的层次聚类解决方法，对应的，也有“自底向上”的。其实可以简单的理解，“自顶向下”就是一步步的细化分组，而“自底向上”就是一步步的归并分组。

**簇数目固定的还是无限制的聚类**

这个属性很好理解，就是你的聚类问题是在执行聚类算法前已经确定聚类的结果应该得到多少簇，还是根据数据本身的特征，由聚类算法选择合适的簇的数目。

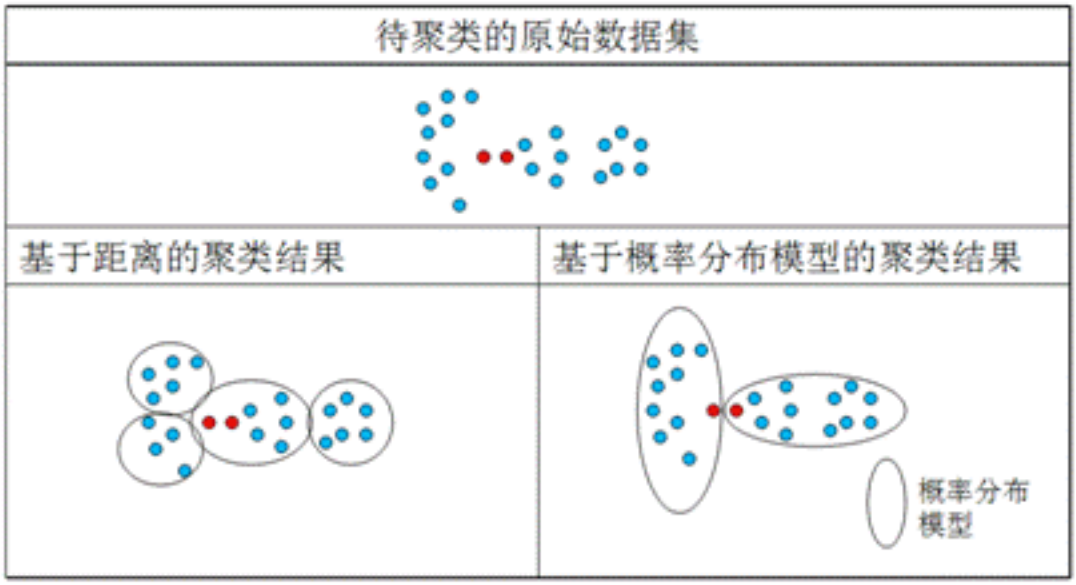
**基于距离还是基于概率分布模型**

在本系列的第二篇介绍协同过滤的文章中，我们已经详细介绍了相似性和距离的概念。基于距离的聚类问题应该很好理解，就是将距离近的相似的对象聚在一起。相比起来，基于概率分布模型的，可能不太好理解，那么下面给个简单的例子。

一个概率分布模型可以理解是在 N 维空间的一组点的分布，而它们的分布往往符合一定的特征，比如组成一个特定的形状。基于概率分布模型的聚类问题，就是在一组对象中，找到能符合特定分布模型的点的集合，他们不一定是距离最近的或者最相似的，

而是能完美的呈现出概率分布模型所描述的模型。
下面图 1 给出了一个例子，对同样一组点集，应用不同的聚类策略，得到完全不同的聚类结果。左侧给出的结果是基于距离的，核心的原则就是将距离近的点聚在一起，右侧给出的基于概率分布模型的聚类结果，这里采用的概率分布模型是一定弧度的椭圆。图中专门标出了两个红色的点，这两点的距离很近，在基于距离的聚类中，将他们聚在一个类中，但基于概率分布模型的聚类则将它们分在不同的类中，只是为了满足特定的概率分布模型（当然这里我特意举了一个比较极端的例子）。所以我们可以看出，在基于概率分布模型的聚类方法里，核心是模型的定义，不同的模型可能导致完全不同的聚类结果。

**图 1 基于距离和基于概率分布模型的聚类问题**




回页首

## Apache Mahout 中的聚类分析框架

Apache Mahout 是 Apache Software Foundation (ASF) 旗下的一个开源项目，提供一些可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序，并且，在 Mahout 的最近版本中还加入了对 Apache Hadoop 的支持，使这些算法可以更高效的运行在云计算环境中。
关于 Apache Mahout 的安装和配置请参考《基于 Apache Mahout 构建社会化推荐引擎》，它是笔者 09 年发表的一篇关于基于 Mahout 实现推荐引擎的 developerWorks 文章，其中详细介绍了 Mahout 的安装步骤。
Mahout 中提供了常用的多种聚类算法，涉及我们刚刚讨论过的各种类型算法的具体实现，下面我们就进一步深入几个典型的聚类算法的原理，优缺点和实用场景，以及如何使用 Mahout 高效的实现它们。

回页首

## 深入聚类算法

深入介绍聚类算法之前，这里先对 Mahout 中对各种聚类问题的数据模型进行简要的介绍。

### 数据模型

Mahout 的聚类算法将对象表示成一种简单的数据模型：向量 (Vector)。在向量数据描述的基础上，我们可以轻松的计算两个对象的相似性，关于向量和向量的相似度计算，本系列的上一篇介绍协同过滤算法的文章中已经进行了详细的介绍，请参考《“探索推荐引擎内部的秘密”系列 - Part 2: 深入推荐引擎相关算法 -- 协同过滤》。
Mahout 中的向量 Vector 是一个每个域是浮点数 (double) 的复合对象，最容易联想到的实现就是一个浮点数的数组。但在具体应用由于向量本身数据内容的不同，比如有些向量的值很密集，每个域都有值；有些呢则是很稀疏，可能只有少量域有值，所以 Mahout 提供了多个实现：

1. DenseVector，它的实现就是一个浮点数数组，对向量里所有域都进行存储，适合用于存储密集向量。
2. RandomAccessSparseVector 基于浮点数的 HashMap 实现的，key 是整形 (int) 类型，value 是浮点数 (double) 类型，它只存储向量中不为空的值，并提供随机访问。
3. SequentialAccessVector 实现为整形 (int) 类型和浮点数 (double) 类型的并行数组，它也只存储向量中不为空的值，但只提供顺序访问。

用户可以根据自己算法的需求选择合适的向量实现类，如果算法需要很多随机访问，应该选择 DenseVector 或者 RandomAccessSparseVector，如果大部分都是顺序访问，SequentialAccessVector 的效果应该更好。
介绍了向量的实现，下面我们看看如何将现有的数据建模成向量，术语就是“如何对数据进行向量化”，以便采用 Mahout 的各种高效的聚类算法。

1. 简单的整形或浮点型的数据
2. 这种数据最简单，只要将不同的域存在向量中即可，比如 n 维空间的点，其实本身可以被描述为一个向量。
3. 枚举类型数据
4. 这类数据是对物体的描述，只是取值范围有限。举个例子，假设你有一个苹果信息的数据集，每个苹果的数据包括：大小，重量，颜色等，我们以颜色为例，设苹果的颜色数据包括：红色，黄色和绿色。在对数据进行建模时，我们可以用数字来表示颜色，红色 =1，黄色 =2，绿色 =3，那么大小直径 8cm，重量 0.15kg，颜色是红色的苹果，建模的向量就是 <8, 0.15, 1>。
5. 下面的清单 1 给出了对以上两种数据进行向量化的例子。
6. 
7. 
8. **清单 1. 创建简单的向量**
9. 

```
// 创建一个二维点集的向量组
public static final double[][] points = { { 1, 1 }, { 2, 1 }, { 1, 2 },
{ 2, 2 }, { 3, 3 },  { 8, 8 }, { 9, 8 }, { 8, 9 }, { 9, 9 }, { 5, 5 },
{ 5, 6 }, { 6, 6 }};
public static List<Vector> getPointVectors(double[][] raw) {
        List<Vector> points = new ArrayList<Vector>();
        for (int i = 0; i < raw.length; i++) {
                double[] fr = raw[i];
// 这里选择创建 RandomAccessSparseVector
                Vector vec = new RandomAccessSparseVector(fr.length);
                // 将数据存放在创建的 Vector 中
vec.assign(fr);
                points.add(vec);
        }
        return points;
}

// 创建苹果信息数据的向量组
public static List<Vector> generateAppleData() {
List<Vector> apples = new ArrayList<Vector>();
// 这里创建的是 NamedVector，其实就是在上面几种 Vector 的基础上，
//为每个 Vector 提供一个可读的名字
        NamedVector apple = new NamedVector(new DenseVector(
        new double[] {0.11, 510, 1}),
                "Small round green apple");
        apples.add(apple);
apple = new NamedVector(new DenseVector(new double[] {0.2, 650, 3}),
                "Large oval red apple");
        apples.add(apple);
        apple = new NamedVector(new DenseVector(new double[] {0.09, 630, 1}),
                "Small elongated red apple");
        apples.add(apple);
        apple = new NamedVector(new DenseVector(new double[] {0.25, 590, 3}),
                "Large round yellow apple");
        apples.add(apple);
        apple = new NamedVector(new DenseVector(new double[] {0.18, 520, 2}),
                "Medium oval green apple");
        apples.add(apple);
```

```
        return apples;
    }
```

10.

11. 文本信息

12. 作为聚类算法的主要应用场景 - 文本分类，对文本信息的建模也是一个常见的问题。在信息检索研究领域已经有很好的建模方式，就是信息检索领域中最常用的向量空间模型 (Vector Space Model, VSM)。因为向量空间模型不是本文的重点，这里给一个简要的介绍，有兴趣的朋友可以查阅参考目录中给出的相关文档。

13. 文本的向量空间模型就是将文本信息建模为一个向量，其中每一个域是文本中出现的一个词的权重。关于权重的计算则有很多中：

    - 最简单的莫过于直接计数，就是词在文本里出现的次数。这种方法简单，但是对文本内容描述的不够精确。
    - 词的频率 (Team Frequency, TF)：就是将词在文本中出现的频率作为词的权重。这种方法只是对于直接计数进行了归一化处理，目的是让不同长度的文本模型有统一的取值空间，便于文本相似度的比较，但可以看出，简单计数和词频都不能解决“高频无意义词汇权重大的问题”，也就是说对于英文文本中，“a”，“the”这样高频但无实际意义的词汇并没有进行过滤，这样的文本模型在计算文本相似度时会很不准确。
    - 词频 - 逆向文本频率 (Term Frequency – Inverse Document Frequency, TF-IDF)：它是对 TF 方法的一种加强，字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在所有文本中出现的频率成反比下降。举个例子，对于“高频无意义词汇”，因为它们大部分会出现在所有的文本中，所以它们的权重会大打折扣，这样就使得文本模型在描述文本特征上更加精确。在信息检索领域，TF-IDF 是对文本信息建模的最常用的方法。

14. 对于文本信息的向量化，Mahout 已经提供了工具类，它基于 Lucene 给出了对文本信息进行分析，然后创建文本向量。下面的清单 2 给出了一个例子，分析的文本数据是路透提供的新闻数据，参考资源里给出了下载地址。将数据集下载后，放在“clustering/reuters”目录下。

15.

16.

17. **清单 2. 创建文本信息的向量**

18.

```
public static void documentVectorize(String[] args) throws Exception{
        //1. 将路透的数据解压缩，Mahout 提供了专门的方法
DocumentClustering.extractReuters();
//2. 将数据存储成 SequenceFile，因为这些工具类就是在 Hadoop 的基础上做的，所以首先我们需要将数据写
//    成 SequenceFile，以便读取和计算
        DocumentClustering.transformToSequenceFile();
//3. 将 SequenceFile 文件中的数据，基于 Lucene 的工具进行向量化
        DocumentClustering.transformToVector();
}

public static void extractReuters(){
//ExtractReuters 是基于 Hadoop 的实现，所以需要将输入输出的文件目录传给它，这里我们可以直接把它映
// 射到我们本地的一个文件夹，解压后的数据将写入输出目录下
        File inputFolder = new File("clustering/reuters");
        File outputFolder = new File("clustering/reuters-extracted");
        ExtractReuters extractor = new ExtractReuters(inputFolder, outputFolder);
extractor.extract();
}

public static void transformToSequenceFile(){
//SequenceFilesFromDirectory 实现将某个文件目录下的所有文件写入一个 SequenceFiles 的功能
// 它其实本身是一个工具类，可以直接用命令行调用，这里直接调用了它的 main 方法
        String[] args = {"-c", "UTF-8", "-i", "clustering/reuters-extracted/", "-o",
        "clustering/reuters-seqfiles"};
```

```
        // 解释一下参数的意义：
//      -c: 指定文件的编码形式，这里用的是"UTF-8"
//      -i: 指定输入的文件目录，这里指到我们刚刚导出文件的目录
//      -o: 指定输出的文件目录

        try {
                SequenceFilesFromDirectory.main(args);
        } catch (Exception e) {
                e.printStackTrace();
        }
}

public static void transformToVector(){
//SparseVectorsFromSequenceFiles 实现将 SequenceFiles 中的数据进行向量化。
// 它其实本身是一个工具类，可以直接用命令行调用，这里直接调用了它的 main 方法
String[] args = {"-i", "clustering/reuters-seqfiles/", "-o",
"clustering/reuters-vectors-bigram", "-a",
"org.apache.lucene.analysis.WhitespaceAnalyzer"
, "-chunk", "200", "-wt", "tfidf", "-s", "5",
"-md", "3", "-x", "90", "-ng", "2", "-ml", "50", "-seq"};
// 解释一下参数的意义：
//      -i: 指定输入的文件目录，这里指到我们刚刚生成 SequenceFiles 的目录
//      -o: 指定输出的文件目录
//      -a: 指定使用的 Analyzer，这里用的是 lucene 的空格分词的 Analyzer
//      -chunk: 指定 Chunk 的大小，单位是 M。对于大的文件集合，我们不能一次 load 所有文件，所以需要
//              对数据进行切块
//      -wt: 指定分析时采用的计算权重的模式，这里选了 tfidf
//      -s:  指定词语在整个文本集合出现的最低频度，低于这个频度的词汇将被丢掉
//      -md: 指定词语在多少不同的文本中出现的最低值，低于这个值的词汇将被丢掉
//      -x:  指定高频词汇和无意义词汇（例如 is，a，the 等）的出现频率上限，高于上限的将被丢掉
//      -ng: 指定分词后考虑词汇的最大长度，例如 1-gram 就是，coca，cola，这是两个词，
//          2-gram 时，coca cola 是一个词汇，2-gram 比 1-gram 在一定情况下分析的更准确。
//      -ml: 指定判断相邻词语是不是属于一个词汇的相似度阈值，当选择 >1-gram 时才有用，其实计算的是
//          Minimum Log Likelihood Ratio 的阈值
//      -seq: 指定生成的向量是 SequentialAccessSparseVectors，没设置时默认生成还是
//     RandomAccessSparseVectors

        try {
                SparseVectorsFromSequenceFiles.main(args);
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

19.

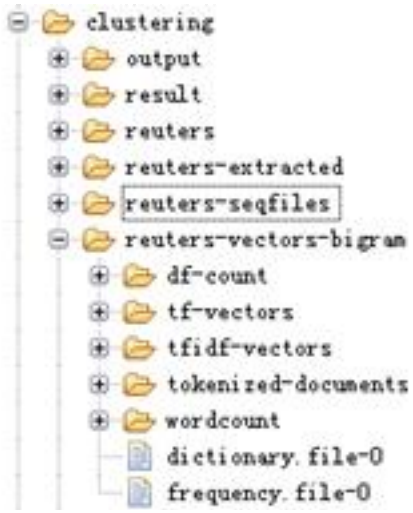
20.

21. 这里补充一点，生成的向量化文件的目录结构是这样的：

22.

23.

24. **图 2 文本信息向量化**

25. 

26.

- df-count 目录：保存着文本的频率信息
- tf-vectors 目录：保存着以 TF 作为权值的文本向量
- tfidf-vectors 目录：保存着以 TFIDF 作为权值的文本向量
- tokenized-documents 目录：保存着分词过后的文本信息
- wordcount 目录：保存着全局的词汇出现的次数
- dictionary.file-0 目录：保存着这些文本的词汇表
- frequcency-file-0 目录：保存着词汇表对应的频率信息。

介绍完向量化问题，下面我们深入分析各个聚类算法，首先介绍的是最经典的 K 均值算法。

## K 均值聚类算法

K 均值是典型的基于距离的排他的划分方法：给定一个 n 个对象的数据集，它可以构建数据的 k 个划分，每个划分就是一个聚类，并且 k<=n，同时还需要满足两个要求：

- 每个组至少包含一个对象
- 每个对象必须属于且仅属于一个组。

K 均值的基本原理是这样的，给定 k，即要构建的划分的数目，

1. 首先创建一个初始划分，随机地选择 k 个对象，每个对象初始地代表了一个簇中心。对于其他的对象，根据其与各个簇中心的距离，将它们赋给最近的簇。
2. 然后采用一种迭代的重定位技术，尝试通过对象在划分间移动来改进划分。所谓重定位技术，就是当有新的对象加入簇或者已有对象离开簇的时候，重新计算簇的平均值，然后对对象进行重新分配。这个过程不断重复，直到没有簇中对象的变化。

当结果簇是密集的，而且簇和簇之间的区别比较明显时，K 均值的效果比较好。对于处理大数据集，这个算法是相对可伸缩的和高效的，它的复杂度是 O(nkt)，n 是对象的个数，k 是簇的数目，t 是迭代的次数，通常 k<<n，且 t<<n，所以算法经常以局部最优结束。

K 均值的最大问题是要求用户必须事先给出 k 的个数，k 的选择一般都基于一些经验值和多次实验结果，对于不同的数据集，k 的取值没有可借鉴性。另外，K 均值对“噪音”和孤立点数据是敏感的，少量这类的数据就能对平均值造成极大的影响。

说了这么多理论的原理，下面我们基于 Mahout 实现一个简单的 K 均值算法的例子。如前面介绍的，Mahout 提供了基本的基于内存的实现和基于 Hadoop 的 Map/Reduce 的实现，分别是 KMeansClusterer 和 KMeansDriver，下面给出一个简单的例子，就基于我们在清单 1 里定义的二维点集数据。

**清单 3. K 均值聚类算法示例**

```
// 基于内存的 K 均值聚类算法实现
public static void kMeansClusterInMemoryKMeans(){
// 指定需要聚类的个数，这里选择 2 类
```

```
int k = 2;
// 指定 K 均值聚类算法的最大迭代次数
int maxIter = 3;
// 指定 K 均值聚类算法的最大距离阈值
double distanceThreshold = 0.01;
// 声明一个计算距离的方法，这里选择了欧几里德距离
DistanceMeasure measure = new EuclideanDistanceMeasure();
// 这里构建向量集，使用的是清单 1 里的二维点集
List<Vector> pointVectors = SimpleDataSet.getPointVectors(SimpleDataSet.points);
// 从点集向量中随机的选择 k 个作为簇的中心
List<Vector> randomPoints = RandomSeedGenerator.chooseRandomPoints(pointVectors, k);
// 基于前面选中的中心构建簇
List<Cluster> clusters = new ArrayList<Cluster>();
int clusterId = 0;
for(Vector v : randomPoints){
        clusters.add(new Cluster(v, clusterId ++, measure));
}
// 调用 KMeansClusterer.clusterPoints 方法执行 K 均值聚类
List<List<Cluster>> finalClusters = KMeansClusterer.clusterPoints(pointVectors,
clusters, measure, maxIter, distanceThreshold);

// 打印最终的聚类结果
for(Cluster cluster : finalClusters.get(finalClusters.size() -1)){
        System.out.println("Cluster id: " + cluster.getId() +
" center: " + cluster.getCenter().asFormatString());
        System.out.println("       Points: " + cluster.getNumPoints());
}
}
// 基于 Hadoop 的 K 均值聚类算法实现
public static void kMeansClusterUsingMapReduce () throws Exception{
// 声明一个计算距离的方法，这里选择了欧几里德距离
        DistanceMeasure measure = new EuclideanDistanceMeasure();
        // 指定输入路径，如前面介绍的一样，基于 Hadoop 的实现就是通过指定输入输出的文件路径来指定数据源的。
        Path testpoints = new Path("testpoints");
        Path output = new Path("output");
        // 清空输入输出路径下的数据
HadoopUtil.overwriteOutput(testpoints);
        HadoopUtil.overwriteOutput(output);
        RandomUtils.useTestSeed();
// 在输入路径下生成点集，与内存的方法不同，这里需要把所有的向量写进文件，下面给出具体的例子
        SimpleDataSet.writePointsToFile(testpoints);
// 指定需要聚类的个数，这里选择 2 类
int k = 2;
// 指定 K 均值聚类算法的最大迭代次数
int maxIter = 3;
        // 指定 K 均值聚类算法的最大距离阈值
double distanceThreshold = 0.01;
// 随机的选择 k 个作为簇的中心
Path clusters = RandomSeedGenerator.buildRandom(testpoints,
new Path(output, "clusters-0"), k, measure);
// 调用 KMeansDriver.runJob 方法执行 K 均值聚类算法
KMeansDriver.runJob(testpoints, clusters, output, measure,
distanceThreshold, maxIter, 1, true, true);
// 调用 ClusterDumper 的 printClusters 方法将聚类结果打印出来。
ClusterDumper clusterDumper = new ClusterDumper(new Path(output,
"clusters-" + maxIter -1), new Path(output, "clusteredPoints"));
```

```
clusterDumper.printClusters(null);
}
//SimpleDataSet 的 writePointsToFile 方法，将测试点集写入文件里
// 首先我们将测试点集包装成 VectorWritable 形式，从而将它们写入文件
public static List<VectorWritable> getPoints(double[][] raw) {
        List<VectorWritable> points = new ArrayList<VectorWritable>();
for (int i = 0; i < raw.length; i++) {
                double[] fr = raw[i];
                Vector vec = new RandomAccessSparseVector(fr.length);
                vec.assign(fr);
// 只是在加入点集前，在 RandomAccessSparseVector 外加了一层 VectorWritable 的包装
                points.add(new VectorWritable(vec));
        }
return points;
}
// 将 VectorWritable 的点集写入文件，这里涉及一些基本的 Hadoop 编程元素，详细的请参阅参考资源里相关的内容
public static void writePointsToFile(Path output) throws IOException {
        // 调用前面的方法生成点集
        List<VectorWritable> pointVectors = getPoints(points);
        // 设置 Hadoop 的基本配置
        Configuration conf = new Configuration();
        // 生成 Hadoop 文件系统对象 FileSystem
        FileSystem fs = FileSystem.get(output.toUri(), conf);
// 生成一个 SequenceFile.Writer，它负责将 Vector 写入文件中
        SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf, output,
        Text.class,  VectorWritable.class);
        // 这里将向量按照文本形式写入文件
        try {
for (VectorWritable vw : pointVectors) {
writer.append(new Text(), vw);
                }
        } finally {
                writer.close();
        }
}
```

执行结果

```
KMeans Clustering In Memory Result
Cluster id: 0
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],\"values\":[1.8,1.8,0.0],\"state\":[1,1,0],
\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,\"highWaterMark\":1,
\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,\"lengthSquared\":-1.0}"}
    Points: 5
Cluster id: 1
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],
\"values\":[7.142857142857143,7.285714285714286,0.0],\"state\":[1,1,0],
\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,\"highWaterMark\":1,
\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,\"lengthSquared\":-1.0}"}
    Points: 7

KMeans Clustering Using Map/Reduce Result
        Weight:  Point:
        1.0: [1.000, 1.000]
        1.0: [2.000, 1.000]
```

```
        1.0: [1.000, 2.000]
        1.0: [2.000, 2.000]
        1.0: [3.000, 3.000]
        Weight:  Point:
        1.0: [8.000, 8.000]
        1.0: [9.000, 8.000]
        1.0: [8.000, 9.000]
        1.0: [9.000, 9.000]
        1.0: [5.000, 5.000]
        1.0: [5.000, 6.000]
        1.0: [6.000, 6.000]
```

介绍完 K 均值聚类算法，我们可以看出它最大的优点是：原理简单，实现起来也相对简单，同时执行效率和对于大数据量的可伸缩性还是较强的。然而缺点也是很明确的，首先它需要用户在执行聚类之前就有明确的聚类个数的设置，这一点是用户在处理大部分问题时都不太可能事先知道的，一般需要通过多次试验找出一个最优的 K 值；其次就是，由于算法在最开始采用随机选择初始聚类中心的方法，所以算法对噪音和孤立点的容忍能力较差。所谓噪音就是待聚类对象中错误的数据，而孤立点是指与其他数据距离较远，相似性较低的数据。对于 K 均值算法，一旦孤立点和噪音在最开始被选作簇中心，对后面整个聚类过程将带来很大的问题，那么我们有什么方法可以先快速找出应该选择多少个簇，同时找到簇的中心，这样可以大大优化 K 均值聚类算法的效率，下面我们就介绍另一个聚类方法：Canopy 聚类算法。

## Canopy 聚类算法

Canopy 聚类算法的基本原则是：首先应用成本低的近似的距离计算方法高效的将数据分为多个组，这里称为一个 Canopy，我们姑且将它翻译为“华盖”，Canopy 之间可以有重叠的部分；然后采用严格的距离计算方式准确的计算在同一 Canopy 中的点，将他们分配与最合适的簇中。Canopy 聚类算法经常用于 K 均值聚类算法的预处理，用来找合适的 k 值和簇中心。

下面详细介绍一下创建 Canopy 的过程：初始，假设我们有一组点集 S，并且预设了两个距离阈值，T1，T2（T1>T2）；然后选择一个点，计算它与 S 中其他点的距离（这里采用成本很低的计算方法），将距离在 T1 以内的放入一个 Canopy 中，同时从 S 中去掉那些与此点距离在 T2 以内的点（这里是为了保证和中心距离在 T2 以内的点不能再作为其他 Canopy 的中心），重复整个过程直到 S 为空为止。

对 K 均值的实现一样，Mahout 也提供了两个 Canopy 聚类的实现，下面我们就看看具体的代码例子。

**清单 4. Canopy 聚类算法示例**

```
//Canopy 聚类算法的内存实现
public static void canopyClusterInMemory () {
        // 设置距离阈值 T1,T2
double T1 = 4.0;
        double T2 = 3.0;
// 调用 CanopyClusterer.createCanopies 方法创建 Canopy，参数分别是：
        //      1. 需要聚类的点集
        //      2. 距离计算方法
        //      3. 距离阈值 T1 和 T2
        List<Canopy> canopies = CanopyClusterer.createCanopies(
SimpleDataSet.getPointVectors(SimpleDataSet.points),
                new EuclideanDistanceMeasure(), T1, T2);
        // 打印创建的 Canopy，因为聚类问题很简单，所以这里没有进行下一步精确的聚类。
        // 有必须的时候，可以拿到 Canopy 聚类的结果作为 K 均值聚类的输入，能更精确更高效的解决聚类问题
for(Canopy canopy：canopies) {
                System.out.println("Cluster id: " + canopy.getId() +
" center: " + canopy.getCenter().asFormatString());
                System.out.println("      Points: " + canopy.getNumPoints());
        }
}

//Canopy 聚类算法的 Hadoop 实现
```

```
public static void canopyClusterUsingMapReduce() throws Exception{
        // 设置距离阈值 T1,T2
double T1 = 4.0;
        double T2 = 3.0;
        // 声明距离计算的方法
        DistanceMeasure measure = new EuclideanDistanceMeasure();
        // 设置输入输出的文件路径
        Path testpoints = new Path("testpoints");
        Path output = new Path("output");
        // 清空输入输出路径下的数据
        HadoopUtil.overwriteOutput(testpoints);
        HadoopUtil.overwriteOutput(output);
        // 将测试点集写入输入目录下
SimpleDataSet.writePointsToFile(testpoints);

// 调用 CanopyDriver.buildClusters 的方法执行 Canopy 聚类，参数是：
        //      1. 输入路径，输出路径
        //      2. 计算距离的方法
        //      3. 距离阈值 T1 和 T2
        new CanopyDriver().buildClusters(testpoints, output, measure, T1, T2, true);
        // 打印 Canopy 聚类的结果
        List<List<Cluster>> clustersM = DisplayClustering.loadClusters(output);
                List<Cluster> clusters = clustersM.get(clustersM.size()-1);
        if(clusters != null){
for(Cluster canopy : clusters) {
   System.out.println("Cluster id: " + canopy.getId() +
" center: " + canopy.getCenter().asFormatString());
  System.out.println("       Points: " + canopy.getNumPoints());
                }
        }
}
```

执行结果

```
Canopy Clustering In Memory Result
Cluster id: 0
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],\"values\":[1.8,1.8,0.0],
\"state\":[1,1,0],\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,
\"highWaterMark\":1,\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},
\"size\":2,\"lengthSquared\":-1.0}"}
     Points: 5
Cluster id: 1
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],\"values\":[7.5,7.666666666666667,0.0],
\"state\":[1,1,0],\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,
\"highWaterMark\":1,\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,
\"lengthSquared\":-1.0}"}
     Points: 6
Cluster id: 2
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],\"values\":[5.0,5.5,0.0],
\"state\":[1,1,0],\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,
\"highWaterMark\":1,\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,
\"lengthSquared\":-1.0}"}
     Points: 2
```

```
Canopy Clustering Using Map/Reduce Result
Cluster id: 0
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],\"values\":[1.8,1.8,0.0],
\"state\":[1,1,0],\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,
\"highWaterMark\":1,\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},
\"size\":2,\"lengthSquared\":-1.0}"}
    Points: 5
Cluster id: 1
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],\"values\":[7.5,7.666666666666667,0.0],
\"state\":[1,1,0],\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,
\"highWaterMark\":1,\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,
\"lengthSquared\":-1.0}"}
    Points: 6
Cluster id: 2
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],
\"values\":[5.333333333333333,5.666666666666667,0.0],\"state\":[1,1,0],
\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,\"highWaterMark\":1,
\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,\"lengthSquared\":-1.0}"}
    Points: 3
```

### 模糊 K 均值聚类算法

模糊 K 均值聚类算法是 K 均值聚类的扩展，它的基本原理和 K 均值一样，只是它的聚类结果允许存在对象属于多个簇，也就是说：它属于我们前面介绍过的可重叠聚类算法。为了深入理解模糊 K 均值和 K 均值的区别，这里我们得花些时间了解一个概念：模糊参数（Fuzziness Factor）。

与 K 均值聚类原理类似，模糊 K 均值也是在待聚类对象向量集合上循环，但是它并不是将向量分配给距离最近的簇，而是计算向量与各个簇的相关性（Association）。假设有一个向量 v，有 k 个簇，v 到 k 个簇中心的距离分别是 $d_1$，$d_2$... $d_k$，那么 V 到第一个簇的相关性 $u_1$可以通过下面的算式计算：

$$u_1 = \frac{1}{\left(\frac{d_1}{d_1}\right)^{\frac{2}{m-1}} + \left(\frac{d_1}{d_2}\right)^{\frac{2}{m-1}} + \ldots + \left(\frac{d_1}{d_k}\right)^{\frac{2}{m-1}}}$$

计算 v 到其他簇的相关性只需将 $d_1$替换为对应的距离。

从上面的算式，我们看出，当 m 近似 2 时，相关性近似 1；当 m 近似 1 时，相关性近似于到该簇的距离，所以 m 的取值在（1，2）区间内，当 m 越大，模糊程度越大，m 就是我们刚刚提到的模糊参数。

讲了这么多理论的原理，下面我们看看如何使用 Mahout 实现模糊 K 均值聚类，同前面的方法一样，Mahout 一样提供了基于内存和基于 Hadoop Map/Reduce 的两种实现 FuzzyKMeansClusterer 和 FuzzyMeansDriver，分别是清单 5 给出了一个例子。

**清单 5. 模糊 K 均值聚类算法示例**

```
public static void fuzzyKMeansClusterInMemory() {
// 指定聚类的个数
int k = 2;
// 指定 K 均值聚类算法的最大迭代次数
int maxIter = 3;
```

```
// 指定 K 均值聚类算法的最大距离阈值
double distanceThreshold = 0.01;
// 指定模糊 K 均值聚类算法的模糊参数
float fuzzificationFactor = 10;
// 声明一个计算距离的方法，这里选择了欧几里德距离
DistanceMeasure measure = new EuclideanDistanceMeasure();
// 构建向量集，使用的是清单 1 里的二维点集
        List<Vector> pointVectors = SimpleDataSet.getPointVectors(SimpleDataSet.points);
// 从点集向量中随机的选择 k 个作为簇的中心
        List<Vector> randomPoints = RandomSeedGenerator.chooseRandomPoints(points, k);
        // 构建初始簇，这里与 K 均值不同，使用了 SoftCluster，表示簇是可重叠的
        List<SoftCluster> clusters = new ArrayList<SoftCluster>();
        int clusterId = 0;
        for (Vector v : randomPoints) {
                clusters.add(new SoftCluster(v, clusterId++, measure));
        }
// 调用 FuzzyKMeansClusterer 的 clusterPoints 方法进行模糊 K 均值聚类
        List<List<SoftCluster>> finalClusters =
        FuzzyKMeansClusterer.clusterPoints(points,
clusters, measure, distanceThreshold, maxIter, fuzzificationFactor);
        // 打印聚类结果
        for(SoftCluster cluster : finalClusters.get(finalClusters.size() - 1)) {
                System.out.println("Fuzzy Cluster id: " + cluster.getId() +
" center: " + cluster.getCenter().asFormatString());
        }
}

public class fuzzyKMeansClusterUsingMapReduce {
// 指定模糊 K 均值聚类算法的模糊参数
        float fuzzificationFactor = 2.0f;
// 指定需要聚类的个数，这里选择 2 类
        int k = 2;
// 指定最大迭代次数
        int maxIter = 3;
// 指定最大距离阈值
        double distanceThreshold = 0.01;
// 声明一个计算距离的方法，这里选择了欧几里德距离
        DistanceMeasure measure = new EuclideanDistanceMeasure();
// 设置输入输出的文件路径
        Path testpoints = new Path("testpoints");
        Path output = new Path("output");
// 清空输入输出路径下的数据
        HadoopUtil.overwriteOutput(testpoints);
        HadoopUtil.overwriteOutput(output);
// 将测试点集写入输入目录下
        SimpleDataSet.writePointsToFile(testpoints);
// 随机的选择 k 个作为簇的中心
        Path clusters = RandomSeedGenerator.buildRandom(testpoints,
new Path(output, "clusters-0"), k, measure);
        FuzzyKMeansDriver.runJob(testpoints, clusters, output, measure, 0.5, maxIter, 1,
fuzzificationFactor, true, true, distanceThreshold, true);
// 打印模糊 K 均值聚类的结果
        ClusterDumper clusterDumper = new ClusterDumper(new Path(output, "clusters-" +
maxIter ),new Path(output, "clusteredPoints"));
        clusterDumper.printClusters(null);
}
```

执行结果

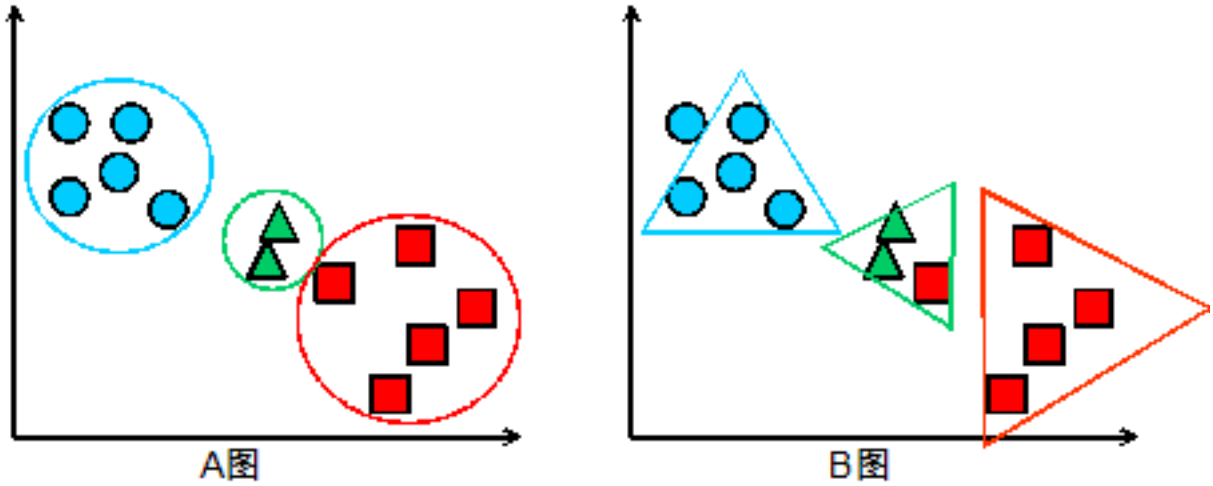
```
Fuzzy KMeans Clustering In Memory Result
Fuzzy Cluster id: 0
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],
\"values\":[1.9750483367699223,1.993870669568863,0.0],\"state\":[1,1,0],
\"freeEntries\":1,\"distinct\":2,\"lowWaterMark\":0,\"highWaterMark\":1,
\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,\"lengthSquared\":-1.0}"}
Fuzzy Cluster id: 1
center:{"class":"org.apache.mahout.math.RandomAccessSparseVector",
"vector":"{\"values\":{\"table\":[0,1,0],
\"values\":[7.924827516566109,7.982356511917616,0.0],\"state\":[1,1,0],
\"freeEntries\":1, \"distinct\":2,\"lowWaterMark\":0,\"highWaterMark\":1,
\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,\"lengthSquared\":-1.0}"}

Funzy KMeans Clustering Using Map Reduce Result
Weight:  Point:
        0.9999249428064162: [8.000, 8.000]
        0.9855340718746096: [9.000, 8.000]
        0.9869963781734195: [8.000, 9.000]
        0.9765978701133124: [9.000, 9.000]
        0.6280999013864511: [5.000, 6.000]
        0.7826097471578298: [6.000, 6.000]
        Weight:  Point:
        0.9672607354172386: [1.000, 1.000]
        0.9794914088151625: [2.000, 1.000]
        0.9803932521191389: [1.000, 2.000]
        0.9977806183197744: [2.000, 2.000]
        0.9793701109946826: [3.000, 3.000]
        0.5422929338028506: [5.000, 5.000]
```

## 狄利克雷聚类算法

前面介绍的三种聚类算法都是基于划分的，下面我们简要介绍一个基于概率分布模型的聚类算法，狄利克雷聚类（Dirichlet Processes Clustering）。

首先我们先简要介绍一下基于概率分布模型的聚类算法（后面简称基于模型的聚类算法）的原理：首先需要定义一个分布模型，简单的例如：圆形，三角形等，复杂的例如正则分布，泊松分布等；然后按照模型对数据进行分类，将不同的对象加入一个模型，模型会增长或者收缩；每一轮过后需要对模型的各个参数进行重新计算，同时估计对象属于这个模型的概率。所以说，基于模型的聚类算法的核心是定义模型，对于一个聚类问题，模型定义的优劣直接影响了聚类的结果，下面给出一个简单的例子，假设我们的问题是将一些二维的点分成三组，在图中用不同的颜色表示，图 A 是采用圆形模型的聚类结果，图 B 是采用三角形模型的聚类结果。可以看出，圆形模型是一个正确的选择，而三角形模型的结果既有遗漏又有误判，是一个错误的选择。

**图 3 采用不同模型的聚类结果**




Mahout 实现的狄利克雷聚类算法是按照如下过程工作的：首先，我们有一组待聚类的对象和一个分布模型。在 Mahout 中使用 ModelDistribution 生成各种模型。初始状态，我们有一个空的模型，然后尝试将对象加入模型中，然后一步一步计算各个对象属于各个模型的概率。下面清单给出了基于内存实现的狄利克雷聚类算法。

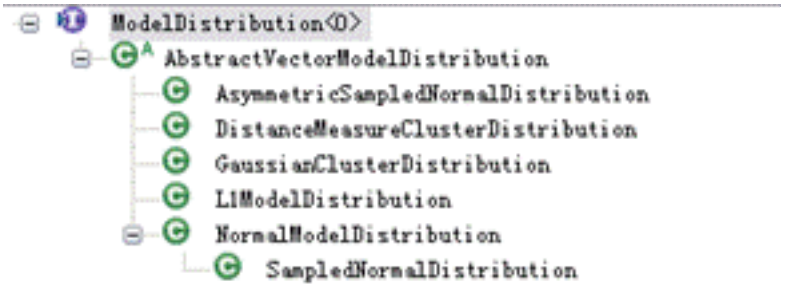
**清单 6. 狄利克雷聚类算法示例**

```
public static void DirichletProcessesClusterInMemory() {
// 指定狄利克雷算法的 alpha 参数，它是一个过渡参数，使得对象分布在不同模型前后能进行光滑的过渡
        double alphaValue = 1.0;
// 指定聚类模型的个数
        int numModels = 3;
// 指定 thin 和 burn 间隔参数，它们是用于降低聚类过程中的内存使用量的
        int thinIntervals = 2;
        int burnIntervals = 2;
// 指定最大迭代次数
        int maxIter = 3;
        List<VectorWritable> pointVectors =
        SimpleDataSet.getPoints(SimpleDataSet.points);
// 初始阶段生成空分布模型，这里用的是 NormalModelDistribution
        ModelDistribution<VectorWritable> model =
new NormalModelDistribution(new VectorWritable(new DenseVector(2)));
// 执行聚类
        DirichletClusterer dc = new DirichletClusterer(pointVectors, model, alphaValue,
numModels, thinIntervals, burnIntervals);
        List<Cluster[]> result = dc.cluster(maxIter);
// 打印聚类结果
        for(Cluster cluster : result.get(result.size() -1)){
                System.out.println("Cluster id: " + cluster.getId() + " center: " +
cluster.getCenter().asFormatString());
                System.out.println("      Points: " + cluster.getNumPoints());
        }
}

执行结果
Dirichlet Processes Clustering In Memory Result
Cluster id: 0
center:{"class":"org.apache.mahout.math.DenseVector",
"vector":"{\"values\":[5.2727272727272725,5.2727272727272725],
\"size\":2,\"lengthSquared\":-1.0}"}
```

```
    Points: 11
Cluster id: 1
center:{"class":"org.apache.mahout.math.DenseVector",
"vector":"{\"values\":[1.0,2.0],\"size\":2,\"lengthSquared\":-1.0}"}
    Points: 1
Cluster id: 2
center:{"class":"org.apache.mahout.math.DenseVector",
"vector":"{\"values\":[9.0,8.0],\"size\":2,\"lengthSquared\":-1.0}"}
    Points: 0
```

Mahout 中提供多种概率分布模型的实现，他们都继承 ModelDistribution，如图 4 所示，用户可以根据自己的数据集的特征选择合适的模型，详细的介绍请参考 Mahout 的官方文档。

**图 4 Mahout 中的概率分布模型层次结构**




### Mahout 聚类算法总结

前面详细介绍了 Mahout 提供的四种聚类算法，这里做一个简要的总结，分析各个算法优缺点，其实，除了这四种以外，Mahout 还提供了一些比较复杂的聚类算法，这里就不一一详细介绍了，详细信息请参考 Mahout Wiki 上给出的聚类算法详细介绍。

**表 1 Mahout 聚类算法总结**

| 算法 | 内存实现 | Map/Reduce 实现 | 簇个数是确定的 | 簇是否允许重叠 |
|---|---|---|---|---|
| K 均值 | KMeansClusterer | KMeansDriver | Y | N |
| Canopy | CanopyClusterer | CanopyDriver | N | N |
| 模糊 K 均值 | FuzzyKMeansClusterer | FuzzyKMeansDriver | Y | Y |
| 狄利克雷 | DirichletClusterer | DirichletDriver | N | Y |

回页首

## 总结

聚类算法被广泛的运用于信息智能处理系统。本文首先简述了聚类概念与聚类算法思想，使得读者整体上了解聚类这一重要的技术。然后从实际构建应用的角度出发，深入的介绍了开源软件 Apache Mahout 中关于聚类的实现框架，包括了其中的数学模型，各种聚类算法以及在不同基础架构上的实现。通过代码示例，读者可以知道针对他的特定的数据问题，怎么样向量化数据，怎么样选择各种不同的聚类算法。

本系列的下一篇将继续深入了解推荐引擎的相关算法 -- 分类。与聚类一样，分类也是一个数据挖掘的经典问题，主要用于提取描述重要数据类的模型，随后我们可以根据这个模型进行预测，推荐就是一种预测的行为。同时聚类和分类往往也是相辅相成的，他们都为在海量数据上进行高效的推荐提供辅助。所以本系列的下一篇文章将详细介绍各类分类算法，它们的原理，优缺点和实用场景，并给出基于 Apache Mahout 的分类算法的高效实现。

最后，感谢大家对本系列的关注和支持。

## 参考资料

**学习**

- 聚类分析：Wikipedia 上关于聚类分析的介绍
- 数据挖掘：概念与技术（韩家伟）：关于数据挖掘的经典著作，详细介绍了数据挖掘领域的各种问题和应用，其中对聚类分析的经典算法也有详尽的讲解。
- 数据挖掘：实用机器学习技术：同样是数据挖掘的经典著作，对领域内的算法，算法的发展进行了详细的介绍。
- “Apache Mahout简介”（Grant Ingersoll，developerWorks，2009 年 10 月）：Mahout 的创始者 Grant Ingersoll 介绍了机器学习的基本概念，并演示了如何使用 Mahout 来实现文档集群、提出建议和组织内容。
- Apache Mahout：Apache Mahout 项目的主页，搜索关于 Mahout 的所有内容。
- Apache Mahout算法总结：Apache Mahout 的 Wiki 上关于实现算法的详细介绍。
- Mahout In Action：Sean Owen 详细介绍了 Mahout 项目，其中有很大篇幅介绍了 Mahout 提供的聚类算法，并给出一些简单的例子。
- TF-IDF：Wikipedia 上关于 TF-IDF 的详细介绍，包括它的计算方法，优缺点，应用场景等。
- 路透数据集：路透提供了大量的新闻数据集，可以作为聚类分析的数据源，本文中对文本聚类分析的部分采用了路透“Reuters-21578”数据集
- Efficient Clustering of High Dimensional Data Sets with Application to Reference Matching，发表于 2000 的 Canopy 算法的论文。
- 狄利克雷分布：Wikipedia 上关于狄利克雷分布的介绍，它是本文介绍的狄利克雷聚类算法的基础
- 基于Apache Mahout构建社会化推荐引擎：笔者 09 年发布的一篇关于基于 Mahout 实现推荐引擎的 developerWorks 文章，其中详细介绍了 Mahout 的安装步骤，并给出一个简单的电影推荐引擎的例子。
- 机器学习：机器学习的 Wikipedia 页面，可帮助您了解关于机器学习的更多信息。
- developerWorks Java技术专区：数百篇关于 Java 编程各个方面的文章。
- developerWorks Web development 专区：通过专门关于 Web 技术的文章和教程，扩展您在网站开发方面的技能。
- developerWorks Ajax 资源中心：这是有关 Ajax 编程模型信息的一站式中心，包括很多文档、教程、论坛、blog、wiki 和新闻。任何 Ajax 的新信息都能在这里找到。
- developerWorks Web 2.0 资源中心，这是有关 Web 2.0 相关信息的一站式中心，包括大量 Web 2.0 技术文章、教程、

下载和相关技术资源。您还可以通过 Web 2.0 新手入门 栏目，迅速了解 Web 2.0 的相关概念。

- 
- 查看 HTML5 专题，了解更多和 HTML5 相关的知识和动向。
- 

**讨论**

- 加入 developerWorks 中文社区。

http://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy3/