“探索推荐引擎内部的秘密”系列将带领读者从浅入深的学习探索推荐引擎的机制，实现方法，其中还涉及一些基本的优化方法，例如聚类和分类的应用。同时在理论讲解的基础上，还会结合 Apache Mahout 介绍如何在大规模数据上实现各种推荐策略，进行策略优化，构建高效的推荐引擎的方法。本文作为这个系列的第一篇文章，将深入介绍推荐引擎的工作原理，和其中涉及的各种推荐机制，以及它们各自的优缺点和适用场景，帮助用户清楚的了解和快速构建适合自己的推荐引擎。

**信息发现**

如今已经进入了一个数据爆炸的时代，随着 Web 2.0 的发展， Web 已经变成数据分享的平台，那么，如何让人们在海量的数据中想要找到他们需要的信息将变得越来越难。

在这样的情形下，搜索引擎（Google，Bing，百度等等）成为大家快速找到目标信息的最好途径。在用户对自己需求相对明确的时候，用搜索引擎很方便的通过关键字搜索很快的找到自己需要的信息。但搜索引擎并不能完全满足用户对信息发现的需求，那是因为在很多情况下，用户其实并不明确自己的需要，或者他们的需求很难用简单的关键字来表述。又或者他们需要更加符合他们个人口味和喜好的结果，因此出现了推荐系统，与搜索引擎对应，大家也习惯称它为推荐引擎。

随着推荐引擎的出现，用户获取信息的方式从简单的目标明确的数据的搜索转换到更高级更符合人们使用习惯的信息发现。

如今，随着推荐技术的不断发展，推荐引擎已经在电子商务 (E-commerce，例如 Amazon，当当网 ) 和一些基于 social 的社会化站点 ( 包括音乐，电影和图书分享，例如豆瓣，Mtime 等 ) 都取得很大的成功。这也进一步的说明了，Web2.0 环境下，在面对海量的数据，用户需要这种更加智能的，更加了解他们需求，口味和喜好的信息发现机制。

[**回页首**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy1/index.html#ibm-pcon)

**推荐引擎**

前面介绍了推荐引擎对于现在的 Web2.0 站点的重要意义，这一章我们将讲讲推荐引擎到底是怎么工作的。推荐引擎利用特殊的信息过滤技术，将不同的物品或内容推荐给可能对它们感兴趣的用户。

**图 1. 推荐引擎工作原理图**

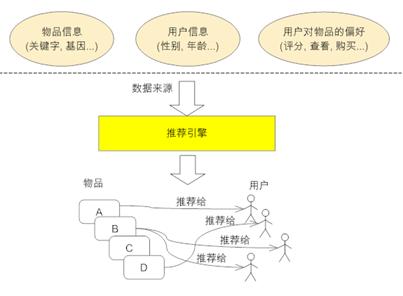


图 1 给出了推荐引擎的工作原理图，这里先将推荐引擎看作黑盒，它接受的输入是推荐的数据源，一般情况下，推荐引擎所需要的数据源包括：

* 要推荐物品或内容的元数据，例如关键字，基因描述等；
* 系统用户的基本信息，例如性别，年龄等
* 用户对物品或者信息的偏好，根据应用本身的不同，可能包括用户对物品的评分，用户查看物品的记录，用户的购买记录等。其实这些用户的偏好信息可以分为两类：
* 显式的用户反馈：这类是用户在网站上自然浏览或者使用网站以外，显式的提供反馈信息，例如用户对物品的评分，或者对物品的评论。
* 隐式的用户反馈：这类是用户在使用网站是产生的数据，隐式的反应了用户对物品的喜好，例如用户购买了某物品，用户查看了某物品的信息等等。

显式的用户反馈能准确的反应用户对物品的真实喜好，但需要用户付出额外的代价，而隐式的用户行为，通过一些分析和处理，也能反映用户的喜好，只是数据不是很精确，有些行为的分析存在较大的噪音。但只要选择正确的行为特征，隐式的用户反馈也能得到很好的效果，只是行为特征的选择可能在不同的应用中有很大的不同，例如在电子商务的网站上，购买行为其实就是一个能很好表现用户喜好的隐式反馈。

推荐引擎根据不同的推荐机制可能用到数据源中的一部分，然后根据这些数据，分析出一定的规则或者直接对用户对其他物品的喜好进行预测计算。这样推荐引擎可以在用户进入的时候给他推荐他可能感兴趣的物品。

[**回页首**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy1/index.html#ibm-pcon)

**推荐引擎的分类**

推荐引擎的分类可以根据很多指标，下面我们一一介绍一下：

1. 推荐引擎是不是为不同的用户推荐不同的数据

根据这个指标，推荐引擎可以分为基于大众行为的推荐引擎和个性化推荐引擎

* + 根据大众行为的推荐引擎，对每个用户都给出同样的推荐，这些推荐可以是静态的由系统管理员人工设定的，或者基于系统所有用户的反馈统计计算出的当下比较流行的物品。
  + 个性化推荐引擎，对不同的用户，根据他们的口味和喜好给出更加精确的推荐，这时，系统需要了解需推荐内容和用户的特质，或者基于社会化网络，通过找到与当前用户相同喜好的用户，实现推荐。

这是一个最基本的推荐引擎分类，其实大部分人们讨论的推荐引擎都是将个性化的推荐引擎，因为从根本上说，只有个性化的推荐引擎才是更加智能的信息发现过程。

1. 根据推荐引擎的数据源

其实这里讲的是如何发现数据的相关性，因为大部分推荐引擎的工作原理还是基于物品或者用户的相似集进行推荐。那么参考图 1 给出的推荐系统原理图，根据不同的数据源发现数据相关性的方法可以分为以下几种：

* + 根据系统用户的基本信息发现用户的相关程度，这种被称为基于人口统计学的推荐（Demographic-based Recommendation）
  + 根据推荐物品或内容的元数据，发现物品或者内容的相关性，这种被称为基于内容的推荐（Content-based Recommendation）
  + 根据用户对物品或者信息的偏好，发现物品或者内容本身的相关性，或者是发现用户的相关性，这种被称为基于协同过滤的推荐（Collaborative Filtering-based Recommendation）。

1. 根据推荐模型的建立方式

可以想象在海量物品和用户的系统中，推荐引擎的计算量是相当大的，要实现实时的推荐务必需要建立一个推荐模型，关于推荐模型的建立方式可以分为以下几种：

* + 基于物品和用户本身的，这种推荐引擎将每个用户和每个物品都当作独立的实体，预测每个用户对于每个物品的喜好程度，这些信息往往是用一个二维矩阵描述的。由于用户感兴趣的物品远远小于总物品的数目，这样的模型导致大量的数据空置，即我们得到的二维矩阵往往是一个很大的稀疏矩阵。同时为了减小计算量，我们可以对物品和用户进行聚类， 然后记录和计算一类用户对一类物品的喜好程度，但这样的模型又会在推荐的准确性上有损失。
  + 基于关联规则的推荐（Rule-based Recommendation）：关联规则的挖掘已经是数据挖掘中的一个经典的问题，主要是挖掘一些数据的依赖关系，典型的场景就是“购物篮问题”，通过关联规则的挖掘，我们可以找到哪些物品经常被同时购买，或者用户购买了一些物品后通常会购买哪些其他的物品，当我们挖掘出这些关联规则之后，我们可以基于这些规则给用户进行推荐。
  + 基于模型的推荐（Model-based Recommendation）：这是一个典型的机器学习的问题，可以将已有的用户喜好信息作为训练样本，训练出一个预测用户喜好的模型，这样以后用户在进入系统，可以基于此模型计算推荐。这种方法的问题在于如何将用户实时或者近期的喜好信息反馈给训练好的模型，从而提高推荐的准确度。

其实在现在的推荐系统中，很少有只使用了一个推荐策略的推荐引擎，一般都是在不同的场景下使用不同的推荐策略从而达到最好的推荐效果，例如 Amazon 的推荐，它将基于用户本身历史购买数据的推荐，和基于用户当前浏览的物品的推荐，以及基于大众喜好的当下比较流行的物品都在不同的区域推荐给用户，让用户可以从全方位的推荐中找到自己真正感兴趣的物品。

[**回页首**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy1/index.html#ibm-pcon)

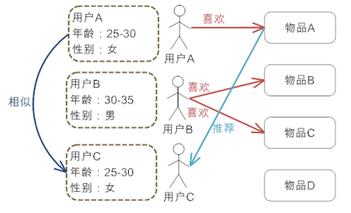
**深入推荐机制**

这一章的篇幅，将详细介绍各个推荐机制的工作原理，它们的优缺点以及应用场景。

**基于人口统计学的推荐**

基于人口统计学的推荐机制（Demographic-based Recommendation）是一种最易于实现的推荐方法，它只是简单的根据系统用户的基本信息发现用户的相关程度，然后将相似用户喜爱的其他物品推荐给当前用户，图 2 给出了这种推荐的工作原理。

**图 2. 基于人口统计学的推荐机制的工作原理**



从图中可以很清楚的看到，首先，系统对每个用户都有一个用户 Profile 的建模，其中包括用户的基本信息，例如用户的年龄，性别等等；然后，系统会根据用户的 Profile 计算用户的相似度，可以看到用户 A 的 Profile 和用户 C 一样，那么系统会认为用户 A 和 C 是相似用户，在推荐引擎中，可以称他们是“邻居”；最后，基于“邻居”用户群的喜好推荐给当前用户一些物品，图中将用户 A 喜欢的物品 A 推荐给用户 C。

这种基于人口统计学的推荐机制的好处在于：

1. 因为不使用当前用户对物品的喜好历史数据，所以对于新用户来讲没有“冷启动（Cold Start）”的问题。
2. 这个方法不依赖于物品本身的数据，所以这个方法在不同物品的领域都可以使用，它是领域独立的（domain-independent）。

那么这个方法的缺点和问题是什么呢？这种基于用户的基本信息对用户进行分类的方法过于粗糙，尤其是对品味要求较高的领域，比如图书，电影和音乐等领域，无法得到很好的推荐效果。可能在一些电子商务的网站中，这个方法可以给出一些简单的推荐。另外一个局限是，这个方法可能涉及到一些与信息发现问题本身无关却比较敏感的信息，比如用户的年龄等，这些用户信息不是很好获取。

**基于内容的推荐**

基于内容的推荐是在推荐引擎出现之初应用最为广泛的推荐机制，它的核心思想是根据推荐物品或内容的元数据，发现物品或者内容的相关性，然后基于用户以往的喜好记录，推荐给用户相似的物品。图 3 给出了基于内容推荐的基本原理。

**图 3. 基于内容推荐机制的基本原理**

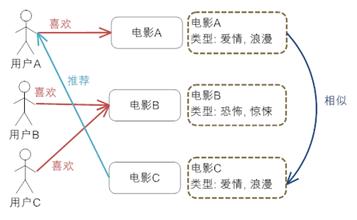


图 3 中给出了基于内容推荐的一个典型的例子，电影推荐系统，首先我们需要对电影的元数据有一个建模，这里只简单的描述了一下电影的类型；然后通过电影的元数据发现电影间的相似度，因为类型都是“爱情，浪漫”电影 A 和 C 被认为是相似的电影（当然，只根据类型是不够的，要得到更好的推荐，我们还可以考虑电影的导演，演员等等）；最后实现推荐，对于用户 A，他喜欢看电影 A，那么系统就可以给他推荐类似的电影 C。

这种基于内容的推荐机制的好处在于它能很好的建模用户的口味，能提供更加精确的推荐。但它也存在以下几个问题：

1. 需要对物品进行分析和建模，推荐的质量依赖于对物品模型的完整和全面程度。在现在的应用中我们可以观察到关键词和标签（Tag）被认为是描述物品元数据的一种简单有效的方法。
2. 物品相似度的分析仅仅依赖于物品本身的特征，这里没有考虑人对物品的态度。
3. 因为需要基于用户以往的喜好历史做出推荐，所以对于新用户有“冷启动”的问题。

虽然这个方法有很多不足和问题，但他还是成功的应用在一些电影，音乐，图书的社交站点，有些站点还请专业的人员对物品进行基因编码，比如潘多拉，在一份报告中说道，在潘多拉的推荐引擎中，每首歌有超过 100 个元数据特征，包括歌曲的风格，年份，演唱者等等。

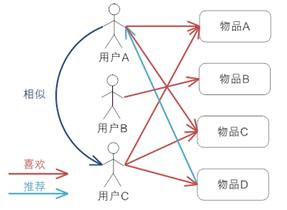
**基于协同过滤的推荐**

随着 Web2.0 的发展，Web 站点更加提倡用户参与和用户贡献，因此基于协同过滤的推荐机制因运而生。它的原理很简单，就是根据用户对物品或者信息的偏好，发现物品或者内容本身的相关性，或者是发现用户的相关性，然后再基于这些关联性进行推荐。基于协同过滤的推荐可以分为三个子类：基于用户的推荐（User-based Recommendation），基于项目的推荐（Item-based Recommendation）和基于模型的推荐（Model-based Recommendation）。下面我们一个一个详细的介绍着三种协同过滤的推荐机制。

**基于用户的协同过滤推荐**

基于用户的协同过滤推荐的基本原理是，根据所有用户对物品或者信息的偏好，发现与当前用户口味和偏好相似的“邻居”用户群，在一般的应用中是采用计算“K- 邻居”的算法；然后，基于这 K 个邻居的历史偏好信息，为当前用户进行推荐。下图 4 给出了原理图。

**图 4. 基于用户的协同过滤推荐机制的基本原理**



上图示意出基于用户的协同过滤推荐机制的基本原理，假设用户 A 喜欢物品 A，物品 C，用户 B 喜欢物品 B，用户 C 喜欢物品 A ，物品 C 和物品 D；从这些用户的历史喜好信息中，我们可以发现用户 A 和用户 C 的口味和偏好是比较类似的，同时用户 C 还喜欢物品 D，那么我们可以推断用户 A 可能也喜欢物品 D，因此可以将物品 D 推荐给用户 A。

基于用户的协同过滤推荐机制和基于人口统计学的推荐机制都是计算用户的相似度，并基于“邻居”用户群计算推荐，但它们所不同的是如何计算用户的相似度，基于人口统计学的机制只考虑用户本身的特征，而基于用户的协同过滤机制可是在用户的历史偏好的数据上计算用户的相似度，它的基本假设是，喜欢类似物品的用户可能有相同或者相似的口味和偏好。

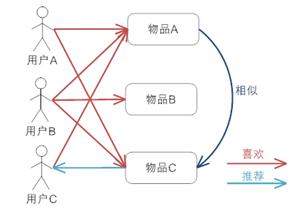
**基于项目的协同过滤推荐**

基于项目的协同过滤推荐的基本原理也是类似的，只是说它使用所有用户对物品或者信息的偏好，发现物品和物品之间的相似度，然后根据用户的历史偏好信息，将类似的物品推荐给用户，图 5 很好的诠释了它的基本原理。

假设用户 A 喜欢物品 A 和物品 C，用户 B 喜欢物品 A，物品 B 和物品 C，用户 C 喜欢物品 A，从这些用户的历史喜好可以分析出物品 A 和物品 C 时比较类似的，喜欢物品 A 的人都喜欢物品 C，基于这个数据可以推断用户 C 很有可能也喜欢物品 C，所以系统会将物品 C 推荐给用户 C。

与上面讲的类似，基于项目的协同过滤推荐和基于内容的推荐其实都是基于物品相似度预测推荐，只是相似度计算的方法不一样，前者是从用户历史的偏好推断，而后者是基于物品本身的属性特征信息。

**图 5. 基于项目的协同过滤推荐机制的基本原理**



同时协同过滤，在基于用户和基于项目两个策略中应该如何选择呢？其实基于项目的协同过滤推荐机制是 Amazon 在基于用户的机制上改良的一种策略，因为在大部分的 Web 站点中，物品的个数是远远小于用户的数量的，而且物品的个数和相似度相对比较稳定，同时基于项目的机制比基于用户的实时性更好一些。但也不是所有的场景都是这样的情况，可以设想一下在一些新闻推荐系统中，也许物品，也就是新闻的个数可能大于用户的个数，而且新闻的更新程度也有很快，所以它的形似度依然不稳定。所以，其实可以看出，推荐策略的选择其实和具体的应用场景有很大的关系。

**基于模型的协同过滤推荐**

基于模型的协同过滤推荐就是基于样本的用户喜好信息，训练一个推荐模型，然后根据实时的用户喜好的信息进行预测，计算推荐。

基于协同过滤的推荐机制是现今应用最为广泛的推荐机制，它有以下几个显著的优点：

1. 它不需要对物品或者用户进行严格的建模，而且不要求物品的描述是机器可理解的，所以这种方法也是领域无关的。
2. 这种方法计算出来的推荐是开放的，可以共用他人的经验，很好的支持用户发现潜在的兴趣偏好

而它也存在以下几个问题：

1. 方法的核心是基于历史数据，所以对新物品和新用户都有“冷启动”的问题。
2. 推荐的效果依赖于用户历史偏好数据的多少和准确性。
3. 在大部分的实现中，用户历史偏好是用稀疏矩阵进行存储的，而稀疏矩阵上的计算有些明显的问题，包括可能少部分人的错误偏好会对推荐的准确度有很大的影响等等。
4. 对于一些特殊品味的用户不能给予很好的推荐。
5. 由于以历史数据为基础，抓取和建模用户的偏好后，很难修改或者根据用户的使用演变，从而导致这个方法不够灵活。

**混合的推荐机制**

在现行的 Web 站点上的推荐往往都不是单纯只采用了某一种推荐的机制和策略，他们往往是将多个方法混合在一起，从而达到更好的推荐效果。关于如何组合各个推荐机制，这里讲几种比较流行的组合方法。

1. 加权的混合（Weighted Hybridization）: 用线性公式（linear formula）将几种不同的推荐按照一定权重组合起来，具体权重的值需要在测试数据集上反复实验，从而达到最好的推荐效果。
2. 切换的混合（Switching Hybridization）：前面也讲到，其实对于不同的情况（数据量，系统运行状况，用户和物品的数目等），推荐策略可能有很大的不同，那么切换的混合方式，就是允许在不同的情况下，选择最为合适的推荐机制计算推荐。
3. 分区的混合（Mixed Hybridization）：采用多种推荐机制，并将不同的推荐结果分不同的区显示给用户。其实，Amazon，当当网等很多电子商务网站都是采用这样的方式，用户可以得到很全面的推荐，也更容易找到他们想要的东西。
4. 分层的混合（Meta-Level Hybridization）: 采用多种推荐机制，并将一个推荐机制的结果作为另一个的输入，从而综合各个推荐机制的优缺点，得到更加准确的推荐。

[**回页首**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy1/index.html#ibm-pcon)

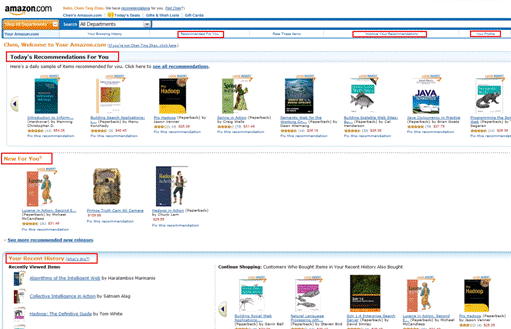
**推荐引擎的应用**

介绍完推荐引擎的基本原理，基本推荐机制，下面简要分析几个有代表性的推荐引擎的应用，这里选择两个领域：Amazon 作为电子商务的代表，豆瓣作为社交网络的代表。

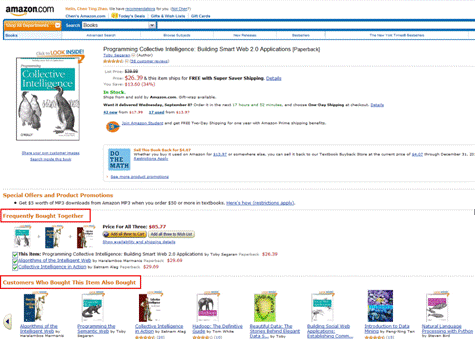
**推荐在电子商务中的应用 – Amazon**

Amazon 作为推荐引擎的鼻祖，它已经将推荐的思想渗透在应用的各个角落。Amazon 推荐的核心是通过数据挖掘算法和比较用户的消费偏好于其他用户进行对比，借以预测用户可能感兴趣的商品。对应于上面介绍的各种推荐机制，Amazon 采用的是分区的混合的机制，并将不同的推荐结果分不同的区显示给用户，图 6 和图 7 展示了用户在 Amazon 上能得到的推荐。

**图 6. Amazon 的推荐机制 - 首页**



**图 7. Amazon 的推荐机制 - 浏览物品**



Amazon 利用可以记录的所有用户在站点上的行为，根据不同数据的特点对它们进行处理，并分成不同区为用户推送推荐：

* 今日推荐 (Today's Recommendation For You): 通常是根据用户的近期的历史购买或者查看记录，并结合时下流行的物品给出一个折中的推荐。
* 新产品的推荐 (New For You): 采用了基于内容的推荐机制 (Content-based Recommendation)，将一些新到物品推荐给用户。在方法选择上由于新物品没有大量的用户喜好信息，所以基于内容的推荐能很好的解决这个“冷启动”的问题。
* 捆绑销售 (Frequently Bought Together): 采用数据挖掘技术对用户的购买行为进行分析，找到经常被一起或同一个人购买的物品集，进行捆绑销售，这是一种典型的基于项目的协同过滤推荐机制。
* 别人购买 / 浏览的商品 (Customers Who Bought/See This Item Also Bought/See): 这也是一个典型的基于项目的协同过滤推荐的应用，通过社会化机制用户能更快更方便的找到自己感兴趣的物品。

值得一提的是，Amazon 在做推荐时，设计和用户体验也做得特别独到：

Amazon 利用有它大量历史数据的优势，量化推荐原因。

* 基于社会化的推荐，Amazon 会给你事实的数据，让用户信服，例如：购买此物品的用户百分之多少也购买了那个物品；
* 基于物品本身的推荐，Amazon 也会列出推荐的理由，例如：因为你的购物框中有 \*\*\*，或者因为你购买过 \*\*\*，所以给你推荐类似的 \*\*\*。

另外，Amazon 很多推荐是基于用户的 profile 计算出来的，用户的 profile 中记录了用户在 Amazon 上的行为，包括看了那些物品，买了那些物品，收藏夹和 wish list 里的物品等等，当然 Amazon 里还集成了评分等其他的用户反馈的方式，它们都是 profile 的一部分，同时，Amazon 提供了让用户自主管理自己 profile 的功能，通过这种方式用户可以更明确的告诉推荐引擎他的品味和意图是什么。

**推荐在社交网站中的应用 – 豆瓣**

豆瓣是国内做的比较成功的社交网站，它以图书，电影，音乐和同城活动为中心，形成一个多元化的社交网络平台，自然推荐的功能是必不可少的，下面我们看看豆瓣是如何推荐的。

**图 8 . 豆瓣的推荐机制 - 豆瓣电影**



当你在豆瓣电影中将一些你看过的或是感兴趣的电影加入你看过和想看的列表里，并为它们做相应的评分，这时豆瓣的推荐引擎已经拿到你的一些偏好信息，那么它将给你展示如图 8 的电影推荐。

**图 9 . 豆瓣的推荐机制 - 基于用户品味的推荐**



豆瓣的推荐是通过“豆瓣猜”，为了让用户清楚这些推荐是如何来的，豆瓣还给出了“豆瓣猜”的一个简要的介绍。

*“你的个人推荐是根据你的收藏和评价自动得出的，每个人的推荐清单都不同。你的收藏和评价越多，豆瓣给你的推荐会越准确和丰富。*  
*每天推荐的内容可能会有变化。随着豆瓣的长大，给你推荐的内容也会越来越准。”*

这一点让我们可以清晰明了的知道，豆瓣必然是基于社会化的协同过滤的推荐，这样用户越多，用户的反馈越多，那么推荐的效果会越来越准确。

相对于 Amazon 的用户行为模型，豆瓣电影的模型更加简单，就是“看过”和“想看”，这也让他们的推荐更加专注于用户的品味，毕竟买东西和看电影的动机还是有很大不同的。

另外，豆瓣也有基于物品本身的推荐，当你查看一些电影的详细信息的时候，他会给你推荐出“喜欢这个电影的人也喜欢的电影”， 如图 10，这是一个基于协同过滤的应用。

**图 10 . 豆瓣的推荐机制 - 基于电影本身的推荐**



[**回页首**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy1/index.html#ibm-pcon)

**总结**

在网络数据爆炸的年代，如何让用户更快的找到想要的数据，如何让用户发现自己潜在的兴趣和需求，无论是对于电子商务还是社会网络的应用都是至关重要的。推荐引擎的出现，使得这个问题越来越被大家关注。但对大多数人来讲，也许还在惊叹它为什么总是能猜到你到底想要些什么。推荐引擎的魔力在于你不清楚在这个推荐背后，引擎到底记录和推理了些什么。

通过这篇综述性的文章，你可以了解，其实推荐引擎只是默默的记录和观察你的一举一动，然后再借由所有用户产生的海量数据分析和发现其中的规律，进而慢慢的了解你，你的需求，你的习惯，并默默的无声息的帮助你快速的解决你的问题，找到你想要的东西。

其实，回头想想，很多时候，推荐引擎比你更了解你自己。

通过第一篇文章，相信大家对推荐引擎有一个清晰的第一印象，本系列的下一篇文章将深入介绍基于协同过滤的推荐策略。在现今的推荐技术和算法中，最被大家广泛认可和采用的就是基于协同过滤的推荐方法。它以其方法模型简单，数据依赖性低，数据方便采集，推荐效果较优等多个优点成为大众眼里的推荐算法“No.1”。本文将带你深入了解协同过滤的秘密，并给出基于 Apache Mahout 的协同过滤算法的高效实现。Apache Mahout 是 ASF 的一个较新的开源项目，它源于 Lucene，构建在 Hadoop 之上，关注海量数据上的机器学习经典算法的高效实现。

**集体智慧和协同过滤**

**什么是集体智慧**

集体智慧 (Collective Intelligence) 并不是 Web2.0 时代特有的，只是在 Web2.0 时代，大家在 Web 应用中利用集体智慧构建更加有趣的应用或者得到更好的用户体验。集体智慧是指在大量的人群的行为和数据中收集答案，帮助你对整个人群得到统计意义上的结论，这些结论是我们在单个个体上无法得到的，它往往是某种趋势或者人群中共性的部分。

Wikipedia 和 Google 是两个典型的利用集体智慧的 Web 2.0 应用：

* Wikipedia 是一个知识管理的百科全书，相对于传统的由领域专家编辑的百科全书，Wikipedia 允许最终用户贡献知识，随着参与人数的增多，Wikipedia 变成了涵盖各个领域的一本无比全面的知识库。也许有人会质疑它的权威性，但如果你从另一个侧面想这个问题，也许就可以迎刃而解。在发行一本书时，作者虽然是权威，但难免还有一些错误，然后通过一版一版的改版，书的内容越来越完善。而在 Wikipedia 上，这种改版和修正被变为每个人都可以做的事情，任何人发现错误或者不完善都可以贡献他们的想法，即便某些信息是错误的，但它一定也会尽快的被其他人纠正过来。从一个宏观的角度看，整个系统在按照一个良性循环的轨迹不断完善，这也正是集体智慧的魅力。
* Google：目前最流行的搜索引擎，与 Wikipedia 不同，它没有要求用户显式的贡献，但仔细想想 Google 最核心的 PageRank 的思想，它利用了 Web 页面之间的关系，将多少其他页面链接到当前页面的数目作为衡量当前页面重要与否的标准；如果这不好理解，那么你可以把它想象成一个选举的过程，每个 Web 页面都是一个投票者同时也是一个被投票者，PageRank 通过一定数目的迭代得到一个相对稳定的评分。Google 其实利用了现在 Internet 上所有 Web 页面上链接的集体智慧，找到哪些页面是重要的。

**什么是协同过滤**

协同过滤是利用集体智慧的一个典型方法。要理解什么是协同过滤 (Collaborative Filtering, 简称 CF)，首先想一个简单的问题，如果你现在想看个电影，但你不知道具体看哪部，你会怎么做？大部分的人会问问周围的朋友，看看最近有什么好看的电影推荐，而我们一般更倾向于从口味比较类似的朋友那里得到推荐。这就是协同过滤的核心思想。

协同过滤一般是在海量的用户中发掘出一小部分和你品位比较类似的，在协同过滤中，这些用户成为邻居，然后根据他们喜欢的其他东西组织成一个排序的目录作为推荐给你。当然其中有一个核心的问题：

* 如何确定一个用户是不是和你有相似的品位？
* 如何将邻居们的喜好组织成一个排序的目录？

协同过滤相对于集体智慧而言，它从一定程度上保留了个体的特征，就是你的品位偏好，所以它更多可以作为个性化推荐的算法思想。可以想象，这种推荐策略在 Web 2.0 的长尾中是很重要的，将大众流行的东西推荐给长尾中的人怎么可能得到好的效果，这也回到推荐系统的一个核心问题：了解你的用户，然后才能给出更好的推荐。

**深入协同过滤的核心**

前面作为背景知识，介绍了集体智慧和协同过滤的基本思想，这一节我们将深入分析协同过滤的原理，介绍基于协同过滤思想的多种推荐机制，优缺点和实用场景。

首先，要实现协同过滤，需要一下几个步骤

* 收集用户偏好
* 找到相似的用户或物品
* 计算推荐

**收集用户偏好**

要从用户的行为和偏好中发现规律，并基于此给予推荐，如何收集用户的偏好信息成为系统推荐效果最基础的决定因素。用户有很多方式向系统提供自己的偏好信息，而且不同的应用也可能大不相同，下面举例进行介绍：

**表 1 用户行为和用户偏好**

| **用户行为** | **类型** | **特征** | **作用** |
| --- | --- | --- | --- |
| 评分 | 显式 | 整数量化的偏好，可能的取值是 [0, n]；n 一般取值为 5 或者是 10 | 通过用户对物品的评分，可以精确的得到用户的偏好 |
| 投票 | 显式 | 布尔量化的偏好，取值是 0 或 1 | 通过用户对物品的投票，可以较精确的得到用户的偏好 |
| 转发 | 显式 | 布尔量化的偏好，取值是 0 或 1 | 通过用户对物品的投票，可以精确的得到用户的偏好。 如果是站内，同时可以推理得到被转发人的偏好（不精确） |
| 保存书签 | 显示 | 布尔量化的偏好，取值是 0 或 1 | 通过用户对物品的投票，可以精确的得到用户的偏好。 |
| 标记标签  (Tag) | 显示 | 一些单词，需要对单词进行分析，得到偏好 | 通过分析用户的标签，可以得到用户对项目的理解，同时可以分析出用户的情感：喜欢还是讨厌 |
| 评论 | 显示 | 一段文字，需要进行文本分析，得到偏好 | 通过分析用户的评论，可以得到用户的情感：喜欢还是讨厌 |
| 点击流  ( 查看 ) | 隐式 | 一组用户的点击，用户对物品感兴趣，需要进行分析，得到偏好 | 用户的点击一定程度上反映了用户的注意力，所以它也可以从一定程度上反映用户的喜好。 |
| 页面停留时间 | 隐式 | 一组时间信息，噪音大，需要进行去噪，分析，得到偏好 | 用户的页面停留时间一定程度上反映了用户的注意力和喜好，但噪音偏大，不好利用。 |
| 购买 | 隐式 | 布尔量化的偏好，取值是 0 或 1 | 用户的购买是很明确的说明这个项目它感兴趣。 |

以上列举的用户行为都是比较通用的，推荐引擎设计人员可以根据自己应用的特点添加特殊的用户行为，并用他们表示用户对物品的喜好。

在一般应用中，我们提取的用户行为一般都多于一种，关于如何组合这些不同的用户行为，基本上有以下两种方式：

* 将不同的行为分组：一般可以分为“查看”和“购买”等等，然后基于不同的行为，计算不同的用户 / 物品相似度。类似于当当网或者 Amazon 给出的“购买了该图书的人还购买了 ...”，“查看了图书的人还查看了 ...”
* 根据不同行为反映用户喜好的程度将它们进行加权，得到用户对于物品的总体喜好。一般来说，显式的用户反馈比隐式的权值大，但比较稀疏，毕竟进行显示反馈的用户是少数；同时相对于“查看”，“购买”行为反映用户喜好的程度更大，但这也因应用而异。

收集了用户行为数据，我们还需要对数据进行一定的预处理，其中最核心的工作就是：减噪和归一化。

* 减噪：用户行为数据是用户在使用应用过程中产生的，它可能存在大量的噪音和用户的误操作，我们可以通过经典的数据挖掘算法过滤掉行为数据中的噪音，这样可以是我们的分析更加精确。
* 归一化：如前面讲到的，在计算用户对物品的喜好程度时，可能需要对不同的行为数据进行加权。但可以想象，不同行为的数据取值可能相差很大，比如，用户的查看数据必然比购买数据大的多，如何将各个行为的数据统一在一个相同的取值范围中，从而使得加权求和得到的总体喜好更加精确，就需要我们进行归一化处理。最简单的归一化处理，就是将各类数据除以此类中的最大值，以保证归一化后的数据取值在 [0,1] 范围中。

进行的预处理后，根据不同应用的行为分析方法，可以选择分组或者加权处理，之后我们可以得到一个用户偏好的二维矩阵，一维是用户列表，另一维是物品列表，值是用户对物品的偏好，一般是 [0,1] 或者 [-1, 1] 的浮点数值。

**找到相似的用户或物品**

当已经对用户行为进行分析得到用户喜好后，我们可以根据用户喜好计算相似用户和物品，然后基于相似用户或者物品进行推荐，这就是最典型的 CF 的两个分支：基于用户的 CF 和基于物品的 CF。这两种方法都需要计算相似度，下面我们先看看最基本的几种计算相似度的方法。

**相似度的计算**

关于相似度的计算，现有的几种基本方法都是基于向量（Vector）的，其实也就是计算两个向量的距离，距离越近相似度越大。在推荐的场景中，在用户 - 物品偏好的二维矩阵中，我们可以将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，或者将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度。下面我们详细介绍几种常用的相似度计算方法：

* 欧几里德距离（Euclidean Distance）

最初用于计算欧几里德空间中两个点的距离，假设 x，y 是 n 维空间的两个点，它们之间的欧几里德距离是：

Figure xxx. Requires a heading

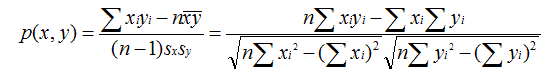
可以看出，当 n=2 时，欧几里德距离就是平面上两个点的距离。

当用欧几里德距离表示相似度，一般采用以下公式进行转换：距离越小，相似度越大

Figure xxx. Requires a heading

* 皮尔逊相关系数（Pearson Correlation Coefficient）

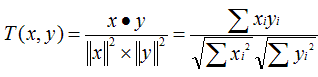
皮尔逊相关系数一般用于计算两个定距变量间联系的紧密程度，它的取值在 [-1，+1] 之间。



sx, sy是 x 和 y 的样品标准偏差。

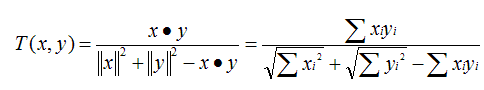
* Cosine 相似度（Cosine Similarity）

Cosine 相似度被广泛应用于计算文档数据的相似度：



* Tanimoto 系数（Tanimoto Coefficient）

Tanimoto 系数也称为 Jaccard 系数，是 Cosine 相似度的扩展，也多用于计算文档数据的相似度：



**相似邻居的计算**

介绍完相似度的计算方法，下面我们看看如何根据相似度找到用户 - 物品的邻居，常用的挑选邻居的原则可以分为两类：图 1 给出了二维平面空间上点集的示意图。

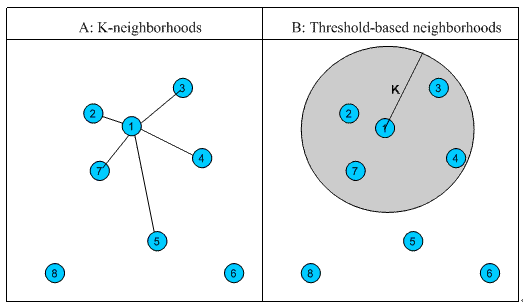
* 固定数量的邻居：K-neighborhoods 或者 Fix-size neighborhoods

不论邻居的“远近”，只取最近的 K 个，作为其邻居。如图 1 中的 A，假设要计算点 1 的 5- 邻居，那么根据点之间的距离，我们取最近的 5 个点，分别是点 2，点 3，点 4，点 7 和点 5。但很明显我们可以看出，这种方法对于孤立点的计算效果不好，因为要取固定个数的邻居，当它附近没有足够多比较相似的点，就被迫取一些不太相似的点作为邻居，这样就影响了邻居相似的程度，比如图 1 中，点 1 和点 5 其实并不是很相似。

* 基于相似度门槛的邻居：Threshold-based neighborhoods

与计算固定数量的邻居的原则不同，基于相似度门槛的邻居计算是对邻居的远近进行最大值的限制，落在以当前点为中心，距离为 K 的区域中的所有点都作为当前点的邻居，这种方法计算得到的邻居个数不确定，但相似度不会出现较大的误差。如图 1 中的 B，从点 1 出发，计算相似度在 K 内的邻居，得到点 2，点 3，点 4 和点 7，这种方法计算出的邻居的相似度程度比前一种优，尤其是对孤立点的处理。

**图 1.相似邻居计算示意图**



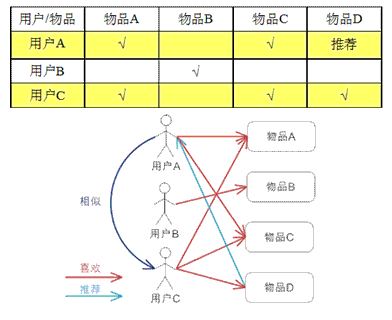
**计算推荐**

经过前期的计算已经得到了相邻用户和相邻物品，下面介绍如何基于这些信息为用户进行推荐。本系列的上一篇综述文章已经简要介绍过基于协同过滤的推荐算法可以分为基于用户的 CF 和基于物品的 CF，下面我们深入这两种方法的计算方法，使用场景和优缺点。

**基于用户的 CF（User CF）**

基于用户的 CF 的基本思想相当简单，基于用户对物品的偏好找到相邻邻居用户，然后将邻居用户喜欢的推荐给当前用户。计算上，就是将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，找到 K 邻居后，根据邻居的相似度权重以及他们对物品的偏好，预测当前用户没有偏好的未涉及物品，计算得到一个排序的物品列表作为推荐。图 2 给出了一个例子，对于用户 A，根据用户的历史偏好，这里只计算得到一个邻居 - 用户 C，然后将用户 C 喜欢的物品 D 推荐给用户 A。

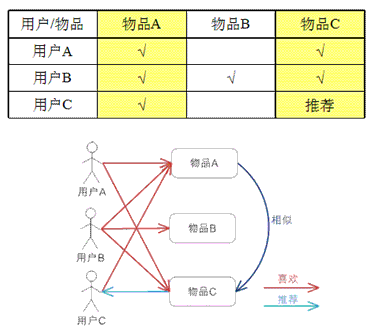
**图 2.基于用户的 CF 的基本原理**



**基于物品的 CF（Item CF）**

基于物品的 CF 的原理和基于用户的 CF 类似，只是在计算邻居时采用物品本身，而不是从用户的角度，即基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似的物品给他。从计算的角度看，就是将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有表示偏好的物品，计算得到一个排序的物品列表作为推荐。图 3 给出了一个例子，对于物品 A，根据所有用户的历史偏好，喜欢物品 A 的用户都喜欢物品 C，得出物品 A 和物品 C 比较相似，而用户 C 喜欢物品 A，那么可以推断出用户 C 可能也喜欢物品 C。

**图 3.基于物品的 CF 的基本原理**



**User CF vs. Item CF**

前面介绍了 User CF 和 Item CF 的基本原理，下面我们分几个不同的角度深入看看它们各自的优缺点和适用场景：

* 计算复杂度

Item CF 和 User CF 是基于协同过滤推荐的两个最基本的算法，User CF 是很早以前就提出来了，Item CF 是从 Amazon 的论文和专利发表之后（2001 年左右）开始流行，大家都觉得 Item CF 从性能和复杂度上比 User CF 更优，其中的一个主要原因就是对于一个在线网站，用户的数量往往大大超过物品的数量，同时物品的数据相对稳定，因此计算物品的相似度不但计算量较小，同时也不必频繁更新。但我们往往忽略了这种情况只适应于提供商品的电子商务网站，对于新闻，博客或者微内容的推荐系统，情况往往是相反的，物品的数量是海量的，同时也是更新频繁的，所以单从复杂度的角度，这两个算法在不同的系统中各有优势，推荐引擎的设计者需要根据自己应用的特点选择更加合适的算法。

* 适用场景

在非社交网络的网站中，内容内在的联系是很重要的推荐原则，它比基于相似用户的推荐原则更加有效。比如在购书网站上，当你看一本书的时候，推荐引擎会给你推荐相关的书籍，这个推荐的重要性远远超过了网站首页对该用户的综合推荐。可以看到，在这种情况下，Item CF 的推荐成为了引导用户浏览的重要手段。同时 Item CF 便于为推荐做出解释，在一个非社交网络的网站中，给某个用户推荐一本书，同时给出的解释是某某和你有相似兴趣的人也看了这本书，这很难让用户信服，因为用户可能根本不认识那个人；但如果解释说是因为这本书和你以前看的某本书相似，用户可能就觉得合理而采纳了此推荐。

相反的，在现今很流行的社交网络站点中，User CF 是一个更不错的选择，User CF 加上社会网络信息，可以增加用户对推荐解释的信服程度。

* 推荐多样性和精度

研究推荐引擎的学者们在相同的数据集合上分别用 User CF 和 Item CF 计算推荐结果，发现推荐列表中，只有 50% 是一样的，还有 50% 完全不同。但是这两个算法确有相似的精度，所以可以说，这两个算法是很互补的。

关于推荐的多样性，有两种度量方法：

第一种度量方法是从单个用户的角度度量，就是说给定一个用户，查看系统给出的推荐列表是否多样，也就是要比较推荐列表中的物品之间两两的相似度，不难想到，对这种度量方法，Item CF 的多样性显然不如 User CF 的好，因为 Item CF 的推荐就是和以前看的东西最相似的。

第二种度量方法是考虑系统的多样性，也被称为覆盖率 (Coverage)，它是指一个推荐系统是否能够提供给所有用户丰富的选择。在这种指标下，Item CF 的多样性要远远好于 User CF, 因为 User CF 总是倾向于推荐热门的，从另一个侧面看，也就是说，Item CF 的推荐有很好的新颖性，很擅长推荐长尾里的物品。所以，尽管大多数情况，Item CF 的精度略小于 User CF， 但如果考虑多样性，Item CF 却比 User CF 好很多。

如果你对推荐的多样性还心存疑惑，那么下面我们再举个实例看看 User CF 和 Item CF 的多样性到底有什么差别。首先，假设每个用户兴趣爱好都是广泛的，喜欢好几个领域的东西，不过每个用户肯定也有一个主要的领域，对这个领域会比其他领域更加关心。给定一个用户，假设他喜欢 3 个领域 A,B,C，A 是他喜欢的主要领域，这个时候我们来看 User CF 和 Item CF 倾向于做出什么推荐：如果用 User CF, 它会将 A,B,C 三个领域中比较热门的东西推荐给用户；而如果用 ItemCF，它会基本上只推荐 A 领域的东西给用户。所以我们看到因为 User CF 只推荐热门的，所以它在推荐长尾里项目方面的能力不足；而 Item CF 只推荐 A 领域给用户，这样他有限的推荐列表中就可能包含了一定数量的不热门的长尾物品，同时 Item CF 的推荐对这个用户而言，显然多样性不足。但是对整个系统而言，因为不同的用户的主要兴趣点不同，所以系统的覆盖率会比较好。

从上面的分析，可以很清晰的看到，这两种推荐都有其合理性，但都不是最好的选择，因此他们的精度也会有损失。其实对这类系统的最好选择是，如果系统给这个用户推荐 30 个物品，既不是每个领域挑选 10 个最热门的给他，也不是推荐 30 个 A 领域的给他，而是比如推荐 15 个 A 领域的给他，剩下的 15 个从 B,C 中选择。所以结合 User CF 和 Item CF 是最优的选择，结合的基本原则就是当采用 Item CF 导致系统对个人推荐的多样性不足时，我们通过加入 User CF 增加个人推荐的多样性，从而提高精度，而当因为采用 User CF 而使系统的整体多样性不足时，我们可以通过加入 Item CF 增加整体的多样性，同样同样可以提高推荐的精度。

* 用户对推荐算法的适应度

前面我们大部分都是从推荐引擎的角度考虑哪个算法更优，但其实我们更多的应该考虑作为推荐引擎的最终使用者 -- 应用用户对推荐算法的适应度。

对于 User CF，推荐的原则是假设用户会喜欢那些和他有相同喜好的用户喜欢的东西，但如果一个用户没有相同喜好的朋友，那 User CF 的算法的效果就会很差，所以一个用户对的 CF 算法的适应度是和他有多少共同喜好用户成正比的。

Item CF 算法也有一个基本假设，就是用户会喜欢和他以前喜欢的东西相似的东西，那么我们可以计算一个用户喜欢的物品的自相似度。一个用户喜欢物品的自相似度大，就说明他喜欢的东西都是比较相似的，也就是说他比较符合 Item CF 方法的基本假设，那么他对 Item CF 的适应度自然比较好；反之，如果自相似度小，就说明这个用户的喜好习惯并不满足 Item CF 方法的基本假设，那么对于这种用户，用 Item CF 方法做出好的推荐的可能性非常低。

通过以上的介绍，相信大家已经对协同过滤推荐的各种方法，原则，特点和适用场景有深入的了解，下面我们就进入实战阶段，重点介绍如何基于 Apache Mahout 实现协同过滤推荐算法。

[**回页首**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy2/#ibm-pcon)

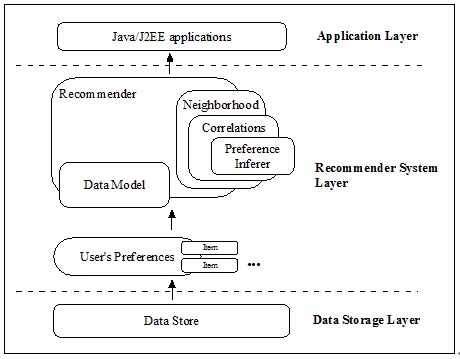
**基于 Apache Mahout 实现高效的协同过滤推荐**

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，并给出一个简单的电影推荐引擎的例子。

Apache Mahout 中提供的一个协同过滤算法的高效实现，它是一个基于 Java 实现的可扩展的，高效的推荐引擎。图 4 给出了 Apache Mahout 中协同过滤推荐实现的组件图，下面我们逐步深入介绍各个部分。

**图 4.组件图**



**数据表示：Data Model**

**Preference**

基于协同过滤的推荐引擎的输入是用户的历史偏好信息，在 Mahout 里它被建模为 Preference（接口），一个 Preference 就是一个简单的三元组 < 用户 ID, 物品 ID, 用户偏好 >，它的实现类是 GenericPreference，可以通过以下语句创建一个 GenericPreference。

GenericPreference preference = new GenericPreference(123, 456, 3.0f);

这其中， 123 是用户 ID，long 型；456 是物品 ID，long 型；3.0f 是用户偏好，float 型。从这个例子我们可以看出，单单一个 GenericPreference 的数据就占用 20 bytes，所以你会发现如果只简单实用数组 Array 加载用户偏好数据，必然占用大量的内存，Mahout 在这方面做了一些优化，它创建了 PreferenceArray（接口）保存一组用户偏好数据，为了优化性能，Mahout 给出了两个实现类，GenericUserPreferenceArray 和 GenericItemPreferenceArray，分别按照用户和物品本身对用户偏好进行组装，这样就可以压缩用户 ID 或者物品 ID 的空间。下面清单 1 的代码以 GenericUserPreferenceArray 为例，展示了如何创建和使用一个 PreferenceArray。

**清单 1. 创建和使用 PreferenceArray**

PreferenceArray userPref = new GenericUserPreferenceArray(2); //size = 2

userPref.setUserID(0, 1L);

userPref.setItemID(0, 101L); //<1L, 101L, 2.0f>

userPref.setValue(0, 2.0f);

userPref.setItemID(1, 102L); //<1L, 102L, 4.0f>

userPref.setValue(1, 4.0f);

Preference pref = userPref.get(1); //<1L, 102L, 4.0f>

为了提高性能 Mahout 还构建了自己的 HashMap 和 Set：FastByIDMap 和 FastIDSet，有兴趣的朋友可以参考 Mahout 官方说明。

**DataModel**

Mahout 的推荐引擎实际接受的输入是 DataModel，它是对用户偏好数据的压缩表示，通过创建内存版 DataModel 的语句我们可以看出：

DataModel model = new GenericDataModel(FastByIDMap<PreferenceArray> map);

他保存在一个按照用户 ID 或者物品 ID 进行散列的 PreferenceArray，而 PreferenceArray 中对应保存着这个用户 ID 或者物品 ID 的所有用户偏好信息。

DataModel 是用户喜好信息的抽象接口，它的具体实现支持从任意类型的数据源抽取用户喜好信息，具体实现包括内存版的 GenericDataModel，支持文件读取的 FileDataModel 和支持数据库读取的 JDBCDataModel，下面我们看看如何创建各种 DataModel。

**清单 2. 创建各种 DataModel**

//In-memory DataModel - GenericDataModel

FastByIDMap<PreferenceArray> preferences = new FastByIDMap<PreferenceArray>();

PreferenceArray prefsForUser1 = new GenericUserPreferenceArray(10);

prefsForUser1.setUserID(0, 1L);

prefsForUser1.setItemID(0, 101L);

prefsForUser1.setValue(0, 3.0f);

prefsForUser1.setItemID(1, 102L);

prefsForUser1.setValue(1, 4.5f);

… (8 more)

preferences.put(1L, prefsForUser1); //use userID as the key

… (more users)

DataModel model = new GenericDataModel(preferences);

//File-based DataModel - FileDataModel

DataModel dataModel = new FileDataModel(new File("preferences.csv");

//Database-based DataModel - MySQLJDBCDataModel

MysqlDataSource dataSource = new MysqlDataSource();

dataSource.setServerName("my\_user");

dataSource.setUser("my\_password");

dataSource.setPassword("my\_database\_host");

JDBCDataModel dataModel = new MySQLJDBCDataModel(dataSource, "my\_prefs\_table",

"my\_user\_column", "my\_item\_column", "my\_pref\_value\_column");

支持文件读取的 FileDataModel，Mahout 没有对文件的格式做过多的要求，只要文件的内容满足以下格式：

* 每一行包括用户 ID, 物品 ID, 用户偏好
* 逗号隔开或者 Tab 隔开
* \*.zip 和 \*.gz 文件会自动解压缩（Mahout 建议在数据量过大时采用压缩的数据存储）

支持数据库读取的 JDBCDataModel，Mahout 提供一个默认的 MySQL 的支持，它对用户偏好数据的存放有以下简单的要求：

* 用户 ID 列需要是 BIGINT 而且非空
* 物品 ID 列需要是 BIGINT 而且非空
* 用户偏好列需要是 FLOAT

建议在用户 ID 和物品 ID 上建索引。

**实现推荐：Recommender**

介绍完数据表示模型，下面介绍 Mahout 提供的协同过滤的推荐策略，这里我们选择其中最经典的三种，User CF, Item CF 和 Slope One。

**User CF**

前面已经详细介绍了 User CF 的原理，这里我们着重看怎么基于 Mahout 实现 User CF 的推荐策略，我们还是从一个例子入手：

**清单 3. 基于 Mahout 实现 User CF**

DataModel model = new FileDataModel(new File("preferences.dat"));

UserSimilarity similarity = new PearsonCorrelationSimilarity(model);

UserNeighborhood neighborhood = new NearestNUserNeighborhood(100, similarity, model);

Recommender recommender = new GenericUserBasedRecommender(model,

neighborhood, similarity);

1. 从文件建立 DataModel，我们采用前面介绍的 FileDataModel，这里假设用户的喜好信息存放在 preferences.dat 文件中。
2. 基于用户偏好数据计算用户的相似度，清单中采用的是 PearsonCorrelationSimilarity，前面章节曾详细介绍了各种计算相似度的方法，Mahout 中提供了基本的相似度的计算，它们都 UserSimilarity 这个接口，实现用户相似度的计算，包括下面这些常用的：

* PearsonCorrelationSimilarity：基于皮尔逊相关系数计算相似度
* EuclideanDistanceSimilarity：基于欧几里德距离计算相似度
* TanimotoCoefficientSimilarity：基于 Tanimoto 系数计算相似度
* UncerteredCosineSimilarity：计算 Cosine 相似度

ItemSimilarity 也是类似的：

1. 根据建立的相似度计算方法，找到邻居用户。这里找邻居用户的方法根据前面我们介绍的，也包括两种：“固定数量的邻居”和“相似度门槛邻居”计算方法，Mahout 提供对应的实现：
   * NearestNUserNeighborhood：对每个用户取固定数量 N 的最近邻居
   * ThresholdUserNeighborhood：对每个用户基于一定的限制，取落在相似度门限内的所有用户为邻居。
2. 基于 DataModel，UserNeighborhood 和 UserSimilarity 构建 GenericUserBasedRecommender，实现 User CF 推荐策略。

**Item CF**

了解了 User CF，Mahout Item CF 的实现与 User CF 类似，是基于 ItemSimilarity，下面我们看实现的代码例子，它比 User CF 更简单，因为 Item CF 中并不需要引入邻居的概念：

**清单 4. 基于 Mahout 实现 Item CF**

DataModel model = new FileDataModel(new File("preferences.dat"));

ItemSimilarity similarity = new PearsonCorrelationSimilarity(model);

Recommender recommender = new GenericItemBasedRecommender(model, similarity);

**Slope One**

如前面介绍的，User CF 和 Item CF 是最常用最容易理解的两种 CF 的推荐策略，但在大数据量时，它们的计算量会很大，从而导致推荐效率较差。因此 Mahout 还提供了一种更加轻量级的 CF 推荐策略：Slope One。

Slope One 是有 Daniel Lemire 和 Anna Maclachlan 在 2005 年提出的一种对基于评分的协同过滤推荐引擎的改进方法，下面简单介绍一下它的基本思想。

图 5 给出了例子，假设系统对于物品 A，物品 B 和物品 C 的平均评分分别是 3，4 和 4。基于 Slope One 的方法会得到以下规律：

* 用户对物品 B 的评分 = 用户对物品 A 的评分 + 1
* 用户对物品 B 的评分 = 用户对物品 C 的评分

基于以上的规律，我们可以对用户 A 和用户 B 的打分进行预测：

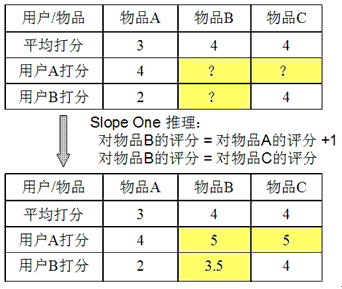
* 对用户 A，他给物品 A 打分 4，那么我们可以推测他对物品 B 的评分是 5，对物品 C 的打分也是 5。
* 对用户 B，他给物品 A 打分 2，给物品 C 打分 4，根据第一条规律，我们可以推断他对物品 B 的评分是 3；而根据第二条规律，推断出评分是 4。当出现冲突时，我们可以对各种规则得到的推断进行就平均，所以给出的推断是 3.5。

这就是 Slope One 推荐的基本原理，它将用户的评分之间的关系看作简单的线性关系：

Y = mX + b;

当 m = 1 时就是 Slope One，也就是我们刚刚展示的例子。

**图 5.Slope One 推荐策略示例**



Slope One 的核心优势是在大规模的数据上，它依然能保证良好的计算速度和推荐效果。Mahout 提供了 Slope One 推荐方法的基本实现，实现代码很简单，参考清单 5.

**清单 5. 基于 Mahout 实现 Slope One**

//In-Memory Recommender

DiffStorage diffStorage = new MemoryDiffStorage(model, Weighting.UNWEIGHTED, false,

Long.MAX\_VALUE));

Recommender recommender = new SlopeOneRecommender(model, Weighting.UNWEIGHTED,

Weighting.UNWEIGHTED, diffStorage);

//Database-based Recommender

AbstractJDBCDataModel model = new MySQLJDBCDataModel();

DiffStorage diffStorage = new MySQLJDBCDiffStorage(model);

Recommender recommender = new SlopeOneRecommender(model, Weighting.WEIGHTED,

Weighting.WEIGHTED, diffStorage);

1. 根据 Data Model 创建数据之间线性关系的模型 DiffStorage。

2. 基于 Data Model 和 DiffStorage 创建 SlopeOneRecommender，实现 Slope One 推荐策略。

[**回页首**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy2/#ibm-pcon)

**总结**

Web2.0 的一个核心思想就是“集体智慧”，基于协同过滤的推荐策略的基本思想就是基于大众行为，为每个用户提供个性化的推荐，从而使用户能更快速更准确的发现所需要的信息。从应用角度分析，现今比较成功的推荐引擎，比如 Amazon，豆瓣，当当等都采用了协同过滤的方式，它不需要对物品或者用户进行严格的建模，而且不要求物品的描述是机器可理解的，是中领域无关的推荐方法，同时这个方法计算出来的推荐是开放的，可以共用他人的经验，很好的支持用户发现潜在的兴趣偏好。基于协同过滤的推荐策略也有不同的分支，它们有不同的实用场景和推荐效果，用户可以根据自己应用的实际情况选择合适的方法，异或组合不同的方法得到更好的推荐效果。

除此之外，本文还介绍了如何基于 Apache Mahout 高效实现协同过滤推荐算法，Apache Mahout 关注海量数据上的机器学习经典算法的高效实现，其中对基于协同过滤的推荐方法也提供了很好的支持，基于 Mahout 你可以轻松的体验高效推荐的神奇。

作为深入推荐引擎相关算法的第一篇文章，本文深入介绍了协同过滤算法，并举例介绍了如何基于 Apache Mahout 高效实现协同过滤推荐算法，Apache Mahout 作为海量数据上的机器学习经典算法的高效实现，其中对基于协同过滤的推荐方法也提供了很好的支持，基于 Mahout 你可以轻松的体验高效推荐的神奇。但我们也发现了在海量数据上高效的运行协同过滤算法以及其他推荐策略这样高复杂的算法还是有很大的挑战的。在面对这个问题的过程中，大家提出了很多减少计算量的方法，而聚类无疑是其中最优的选择。所以本系列的下一篇文章将详细介绍各类聚类算法，它们的原理，优缺点和实用场景，并给出基于 Apache Mahout 的聚类算法的高效实现，并分析在推荐引擎的实现中，如何通过引入聚类来解决大数据量造成的海量计算，从而提供高效的推荐。

# 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

智能推荐大都基于海量数据的计算和处理，然而我们发现在海量数据上高效的运行协同过滤算法以及其他推荐策略这样高复杂的算法是有很大的挑战的，在面对解决这个问题的过程中，大家提出了很多减少计算量的方法，而聚类无疑是其中最优的选择之一。 聚类 (Clustering) 是一个数据挖掘的经典问题，它的目的是将数据分为多个簇 (Cluster)，在同一个簇中的对象之间有较高的相似度，而不同簇的对象差别较大。聚类被广泛的应用于数据处理和统计分析领域。Apache Mahout 是 ASF(Apache Software Foundation) 的一个较新的开源项目，它源于 Lucene，构建在 Hadoop 之上，关注海量数据上的机器学习经典算法的高效实现。本文主要介绍如何基于 Apache Mahout 实现高效的聚类算法，从而实现更高效的数据处理和分析的应用。

**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。

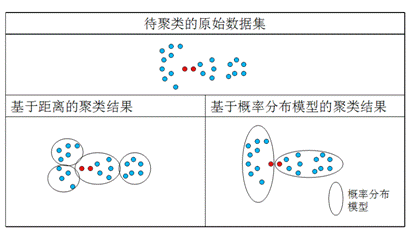
**基于距离还是基于概率分布模型**

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**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy3/#ibm-pcon)

**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**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy3/#ibm-pcon)

**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

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。

1. 枚举类型数据

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>。

下面的清单 1 给出了对以上两种数据进行向量化的例子。

**清单 1. 创建简单的向量**

// 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;

}

1. 文本信息

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。

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。

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。

**清单 2. 创建文本信息的向量**

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) {

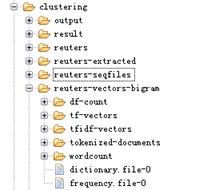
e.printStackTrace();

}

}

这里补充一点，生成的向量化文件的目录结构是这样的：

**图 2 文本信息向量化**



* + 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,

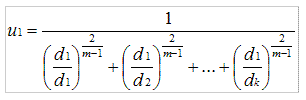
\"minLoadFactor\":0.2,\"maxLoadFactor\":0.5},\"size\":2,\"lengthSquared\":-1.0}"}

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 d1，d2… dk，那么 V 到第一个簇的相关性 u1可以通过下面的算式计算：

****

**计算 v 到其他簇的相关性只需将 d1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]

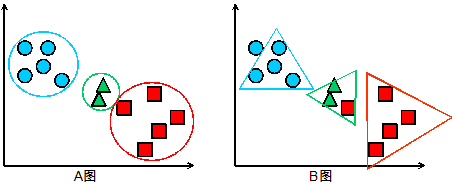
0.5422929338028506: [5.000, 5.000]

**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

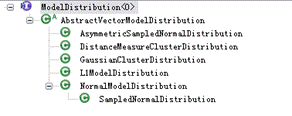
center:{"class":"org.apache.mahout.math.DenseVector",

"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**](https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy3/#ibm-pcon)

**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。