推荐系统概述

1.推荐系统概念

推荐系统是一种帮助用户快速发现有用信息的工具。

和搜索引擎不同的是，推荐系统不需要用户提供明确的需求，而是通过分析用户的历史行为给用户的兴趣建模，从而主动给用户推荐能够满足他们兴趣和需求的信息。因此，从某种意义上说，推荐系统和搜索引擎对于用户来说是两个互补的工具。搜索引擎满足了用户有明确目的时的主动查找需求，而推荐系统能够在用户没有明确目的的时候帮助他们发现感兴趣的新内容。

2.推荐系统应用

购物网站推荐
电影网站推荐
音乐网站推荐
社交网络推荐
个性化广告

3.推荐分类

社会化推荐
基于内容的推荐
基于协同过滤的推荐

4.推荐系统的实现方式

(1)离线实验

离线实验的方法一般由如下几个步骤构成：

通过日志系统获得用户行为数据，并按照一定格式生成一个标准的数据集；
将数据集按照一定的规则分成训练集和测试集；
在训练集上训练用户兴趣模型，在测试集上进行预测；
通过事先定义的离线指标评测算法在测试集上的预测结果。

从上面的步骤可以看到，推荐系统的离线实验都是在数据集上完成的，也就是说它不需要一个实际的系统来供它实验，而只要有一个从实际系统日志中提取的数据集即可。这种实验方法的好处是不需要真实用户参与，可以直接快速地计算出来，从而方便、快速地测试大量不同的算法。

它的主要缺点是无法获得很多商业上关注的指标，如点击率、转化率等，而找到和商业指标非常相关的离线指标也是很困难的事情。

(2)用户调查

用户调查需要有一些真实用户，让他们在需要测试的推荐系统上完成一些任务。在他们完成任务时，我们需要观察和记录他们的行为，并让他们回答一些问题。最后，我们需要通过分析他们的行为和答案了解测试系统的性能。

用户调查也有一些缺点。首先，用户调查成本很高，需要用户花大量时间完成一个个任务，并回答相关的问题。有些时候，还需要花钱雇用测试用户。因此，大多数情况下很难进行大规模的用户调查，而对于参加人数较少的用户调查，得出的很多结论往往没有统计意义。因此，我们在做用户调查时，一方面要控制成本，另一方面又要保证结果的统计意义。

(3)在线实验

在完成离线实验和必要的用户调查后，可以将推荐系统上线做AB测试，将它和旧的算法进行比较。

AB测试是一种很常用的在线评测算法的实验方法。它通过一定的规则将用户随机分成几组，并对不同组的用户采用不同的算法，然后通过统计不同组用户的各种不同的评测指标比较不同算法，比如可以统计不同组用户的点击率，通过点击率比较不同算法的性能。

AB测试的优点是可以公平获得不同算法实际在线时的性能指标，包括商业上关注的指标。AB测试的缺点主要是周期比较长，必须进行长期的实验才能得到可靠的结果。因此一般不会用AB测试测试所有的算法，而只是用它测试那些在离线实验和用户调查中表现很好的算法。

5.推荐系统评测指标

(1)用户满意度

用户作为推荐系统的重要参与者，其满意度是评测推荐系统的最重要指标。但是，用户满意度没有办法离线计算，只能通过用户调查或者在线实验获得。

用户调查获得用户满意度主要是通过调查问卷的形式。用户对推荐系统的满意度分为不同的层次。

(2)预测准确度

预测准确度度量一个推荐系统或者推荐算法预测用户行为的能力。这个指标是最重要的推荐系统离线评测指标。

在计算该指标时需要有一个离线的数据集，该数据集包含用户的历史行为记录。然后，将该数据集通过时间分成训练集和测试集。最后，通过在训练集上建立用户的行为和兴趣模型预测用户在测试集上的行为，并计算预测行为和测试集上实际行为的重合度作为预测准确度。

- 评分预测

  很多提供推荐服务的网站都有一个让用户给物品打分的功能（如图1-26所示）。那么，如果知道了用户对物品的历史评分，就可以从中习得用户的兴趣模型，并预测该用户在将来看到一个他没有评过分的物品时，会给这个物品评多少分。预测用户对物品评分的行为称为评分预测。

  评分预测的预测准确度一般通过均方根误差（RMSE）表示：
  对于测试集中的一个用户u和物品i，令r ui 是用户u对物品i的实际评分，而 ˆ r ui 是推荐算法给出的预测评分

$$\mathrm{RMSE}=\frac{\sqrt{\sum_{u,i\in T}(r_{ui}-\hat{r}_{ui})^2}}{|T|}$$

- TopN 推荐

  网站在提供推荐服务时，一般是给用户一个个性化的推荐列表，这种推荐叫做 TopN 推荐。
  TopN 推荐的预测准确率一般通过准确率（ precision ） / 召回率（ recall ）度量。

  令 R(u) 是根据用户在训练集上的行为给用户作出的推荐列表，而 T(u) 是用户在测试集上的行为列表。那么，推荐结果的召回率定义为：

$$\mathrm{Recall}=\frac{\sum_{u\in U}|R(u)\cap T(u)|}{\sum_{u\in U}|T(u)|}$$

  推荐结果的准确率定义为：

$$\mathrm{Precision}=\frac{\sum_{u\in U}|R(u)\cap T(u)|}{\sum_{u\in U}|R(u)|}$$

(3)覆盖率

覆盖率（ coverage ）描述一个推荐系统对物品长尾的发掘能力。覆盖率有不同的定义方法，最简单的定义为推荐系统能够推荐出来的物品占总物品集合的比例。

推荐系统给每个用户推荐一个长度为 N 的物品列表 R(u) 。那么推荐系统的覆盖率可以通过下面的公式计算：

$$\mathrm{Coverage}=\frac{\left|\bigcup_{u\in U}R(u)\right|}{|I|}$$

(4)多样性

为了满足用户广泛的兴趣，推荐列表需要能够覆盖用户不同的兴趣领域，即推荐结果需要具有多样性。
尽管用户的兴趣在较长的时间跨度中是一样的，但具体到用户访问推荐系统的某一刻，其兴趣往往是单一的，那么如果推荐列表只能覆盖用户的一个兴趣点，而这个兴趣点不是用户这个时刻的兴趣点，推荐列表就不会让用户满意。反

之，如果推荐列表比较多样，覆盖了用户绝大多数的兴趣点，那么就会增加用户找到感兴趣物品的概率。因此给用户的推荐列表也需要满足用户广泛的兴趣，即具有多样性。
假设定义了物品 i 和 j 之间的相似度

$$s(i,j) \in [0,1]$$

那么用户 u 的推荐列表 R(u) 的多样性定义如下

$$\text{Diversity} = 1 - \frac{\sum_{i,j \in R(u), i \neq j} s(i,j)}{\frac{1}{2}\left|R(u)\right|\left(\left|R(u)\right| - 1\right)}$$

而推荐系统的整体多样性可以定义为所有用户推荐列表多样性的平均值：

$$\text{Diversity} = \frac{1}{\left|U\right|}\sum_{u \in U} \text{Diversity}(R(u))$$

(5)新颖性
新颖的推荐是指给用户推荐那些他们以前没有听说过的物品。在一个网站中实现新颖性的最简单办法是，把那些用户之前在网站中对其有过行为的物品从推荐列表中过滤掉。比如在一个视频网站中，新颖的推荐不应该给用户推荐那些他们已经看过、打过分或者浏览过的视频。但是，有些视频可能是用户在别的网站看过，或者是在电视上看过，因此仅仅过滤掉本网站中用户有过行为的物品还不能完全实现新颖性。
因此，要准确地统计新颖性需要做用户调查。

(6)惊喜度

(7)信任度
度量推荐系统的信任度只能通过问卷调查的方式，询问用户是否信任推荐系统的推荐结果。
(8)实时性
推荐系统的实时性包括两个方面。

推荐系统需要实时地更新推荐列表来满足用户新的行为变化。
很多推荐系统都会在离线状态每天计算一次用户推荐列表，然后于在线期间将推荐列表展示给用户。这种设计显然是无法满足实时性的。
与用户行为相应的实时性，可以通过推荐列表的变化速率来评测。

推荐系统需要能够将新加入系统的物品推荐给用户。
这主要考验了推荐系统处理物品冷启动的能力。
可以利用用户推荐列表中有多大比例的物品是当天新加的来评测。

(9)健壮性
绝大部分推荐系统都是通过分析用户的行为实现推荐算法的。那么，我们可以很简单地攻击这个算法，让自己的商品在这个推荐列表中获得比较高的排名，比如可以注册很多账号，用这些账号同时购买 A 和自己的商品。
雇用一批人给自己的商品非常高的评分，而评分行为是推荐系统依赖的重要用户行为。

算法健壮性的评测主要利用模拟攻击。
首先，给定一个数据集和一个算法，可以用这个算法给这个数据集中的用户生成推荐列表。然后，用常用的攻击方法向数据集中注入噪声数据，然后利用算法在注入噪声后的数据集上再次给用户生成推荐列表。最后，通过比较攻击前后推荐列表的相似度评测算法的健壮性。如果攻击后的推荐列表相对于攻击前没有发生大的变化，就说明算法比较健壮

(10)商业目标
设计推荐系统时需要考虑最终的商业目标，而网站使用推荐系统的目的除了满足用户发现内容的需求，也需要利用推荐系统加快实现商业上的指标。

6. 协同过滤实现推荐实现原理

协同过滤算法的离线实验一般如下设计。
首先，将用户行为数据集按照均匀分布随机分成 M份，挑选一份作为测试集，将剩下的 M -1 份作为训练集。然后在训练集上建立用户兴趣模型，并在测试集上对用户行为进行预测，统计出相应的评测指标。为了保证评测指标并不是过拟合的结果，需要进行 M 次实验，并且每次都使用不同的测试集。然后将 M 次实验测出的评测指标的平均值作为最终的评测指标。

协同过滤实现推荐主要有两种方式:

基于用户的协同过滤算法

基于物品的协同过滤算法

7. 基于用户的协同过滤算法 UserCF

基于用户的协同过滤算法主要包括两个步骤

找到和目标用户兴趣相似的用户集合。

找到这个集合中的用户喜欢的，且目标用户没有听说过的物品推荐给目标用户。

(1)找到和目标用户兴趣相似的用户集合

协同过滤算法主要利用行为的相似度计算兴趣的相似度。

给定用户 u 和用户 v，令 N(u) 表示用户 u 曾经有过正反馈的物品集合，令 N(v)为用户 v 曾经有过正反馈的物品集合。

则 u 和 v 的兴趣相似度：

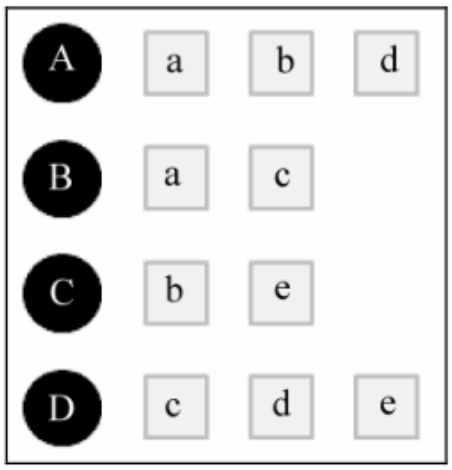
$$w_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$

或

$$w_{uv} = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$$

例中，用户 A 对物品 {a, b, d} 有过行为，用户 B 对物品 {a, c} 有过行为，利用余弦相似度公式计算用户 A 和用户 B 的兴趣相似度为：

$$w_{AB} = \frac{|\{a,b,d\} \cap \{a,c\}|}{\sqrt{|\{a,b,d\}|\ |\{a,c\}|}} = \frac{1}{\sqrt{6}}$$



(2)找到这个集合中的用户喜欢的，且目标用户没有听说过的物品推荐给目标用户

如下的公式度量了 UserCF 算法中用户 u 对物品 i 的感兴趣程度

$$p(u,i) = \sum_{v \in S(u,K) \cap N(i)} w_{uv} r_{vi}$$

其中S(u, K) 包含和用户 u 兴趣最接近的 K 个用户， N(i) 是对物品 i 有过行为的用户集合， w uv是用户 u 和用户 v 的兴趣相似度， r vi 代表用户 v 对物品 i 的兴趣，因为使用的是单一行为的隐反馈数据，所以所有的 r vi =1

8. 基于物品的协同过滤算法 ItemCF

基于物品的协同过滤（ item-based collaborative filtering ）算法是目前业界应用最多的算法

基于物品的协同过滤算法主要包括两个步骤

计算物品之间的相似度。

根据物品的相似度和用户的历史行为给用户生成推荐列表。

(1)计算物品之间的相似度

下面的公式定义物品的相似度：

$$w_{ij} = \frac{|N(i) \cap N(j)|}{|N(i)|}$$

其中分母 |N(i)| 是喜欢物品 i 的用户数，而分子

$$|N(i) \cap N(j)|$$

是同时喜欢物品 i 和物品 j 的用户数

上述公式虽然看起来很有道理，但是却存在一个问题。如果物品 j 很热门，很多人都喜欢，那么 W ij 就会很大，接近 1 。因此，该公式会造成任何物品都会和热门的物品有很大的相似度，这对于致力于挖掘长尾信息的推荐系统来说显然不是一个好的特性。为了避免推荐出热门的物品，

可以用下面的公式：

$$w_{ij} = \frac{|N(i) \cap N(j)|}{\sqrt{|N(i)||N(j)|}}$$

这个公式惩罚了物品 j 的权重，因此减轻了热门物品会和很多物品相似的可能性。

在得到物品之间的相似度后，ItemCF通过如下公式计算用户u对一个物品j的兴趣

$$p_{uj} = \sum_{i \in N(u) \cap S(j,K)} w_{ji} r_{ui}$$

这里N(u)是用户喜欢的物品的集合，S(j,K)是和物品j最相似的K个物品的集合，w ji 是物品j和i的相似度，r ui 是用户u对物品i的兴趣。

9.推荐系统冷启动问题

对于很多做纯粹推荐系统的网站或者很多在开始阶段就希望有个性化推荐应用的网站来说，如何在没有大量用户数据的情况下设计个性化推荐系统并且让用户对推荐结果满意从而愿意使用推荐系统，就是冷启动的问题。

冷启动问题（cold start）主要分3类

用户冷启动

用户冷启动主要解决如何给新用户做个性化推荐的问题。当新用户到来时，我们没有他的行为数据，所以也无法根据他的历史行为预测其兴趣，从而无法借此给他做个性化推荐。

物品冷启动

物品冷启动主要解决如何将新的物品推荐给可能对它感兴趣的用户这一问题。

系统冷启动

系统冷启动主要解决如何在一个新开发的网站上（还没有用户，也没有用户行为，只有一些物品的信息）设计个性化推荐系统，从而在网站刚发布时就让用户体验到个性化推荐服务这一问题。

解决方案：

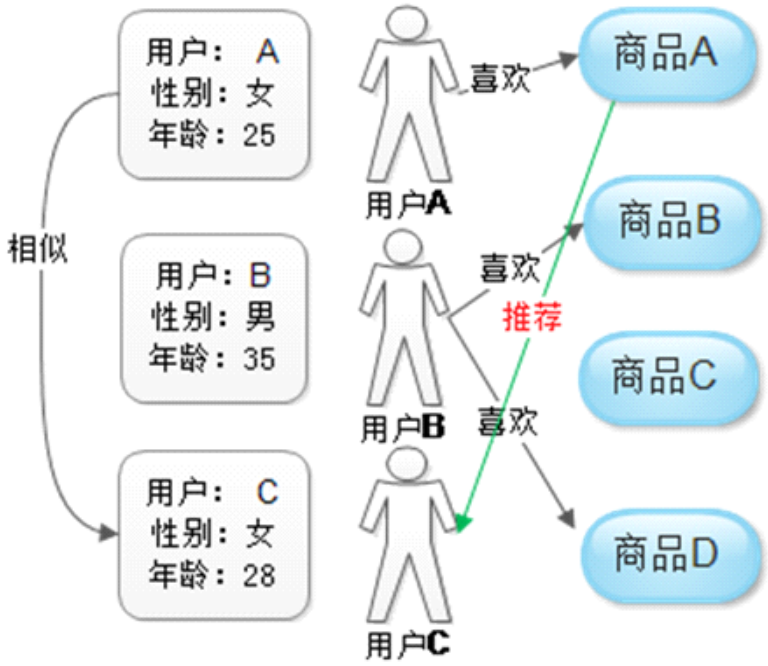
- 提供非个性化的推荐 非个性化推荐的最简单例子就是热门排行榜，我们可以给用户推荐热门排行榜，然后等到用户数据收集到一定的时候，再切换为个性化推荐。
- 利用用户注册时提供的年龄、性别等数据做粗粒度的个性化。
- 利用用户的社交网络账号登录（需要用户授权），导入用户在社交网站上的好友信息，然后给用户推荐其好友喜欢的物品。
- 要求用户在登录时对一些物品进行反馈，收集用户对这些物品的兴趣信息，然后给用户推荐那些和这些物品相似的物品。
- 对于新加入的物品，可以利用内容信息，将它们推荐给喜欢过和它们相似的物品的用户。
- 在系统冷启动时，可以引入专家的知识，通过一定的高效方式迅速建立起物品的相关度表。

Spark实现推荐案例01

协同过滤推荐算法，是最经典、最常用的推荐算法。通过分析用户兴趣，在用户群中找到指定用户的相似用户，综合这些相似用户对某一信息的评价，形成系统关于该指定用户对此信息的喜好程度预测。

要实现协同过滤，需要以下几个步骤：

1）收集用户偏好；
2）找到相似的用户或物品；
3）计算推荐。



用户评分

从用户的行为和偏好中发现规律，并基于此进行推荐，所以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。用户有很多种方式向系统提供自己的偏好信息，比如：评分、投票、转发、保存书签、购买、点击流、页面停留时间等。

1. 将不同的行为分组

一般可以分为查看和购买，然后基于不同的用户行为，计算不同用户或者物品的相似度。

2. 对不同行为进行加权

对不同行为产生的用户喜好进行加权，然后求出用户对物品的总体喜好。当我们收集好用户的行为数据后，还要对 数据进行预处理，最核心的工作就是减噪和归一化。

表 14-1　用户评分表

用户/物品	物品A	物品B	物品C
用户A	0.1	0.8	1
用户B	0.1	0	0.02
用户C	0.5	0.3	0.1

对用户的行为分析得到用户的偏好后，可以根据用户的偏好计算相似用户和物品，然后可以基于相似用户或物品进行推荐。这就是协同过滤中的两个分支了，即基于用户的协同过滤和基于物品的协同过滤。

	物品1	物品2	物品3	物品4	物品5	物品6
用户A	1	1	0	0	0	1
用户B	1	0	1	1	0	0
用户C	1	1	1	0	0	0
用户D	0	0	1	1	1	0
用户E	0	0	0	1	1	1
用户F	1	0	1	0	1	0

1. 同现相似度

物品 i 和物品 j 的同现相似度公式定义：

$$w_{i,j}=\frac{|N(i)\cap N(j)|}{|N(i)|}$$

其中，分母$|N(i)|$是喜欢物品 i 的用户数，而分子$|N(i)\cap N(j)|$是同时喜欢物品 i 和物品 j 的用户数据。因此，上述公式可以理解为喜欢物品 i 的用户中有多少比例的用户也喜欢物品 j。

上述公式存在一个问题，如果物品 j 是热门物品，很多人都喜欢，那么 $w_{i,j}$ 就会很大，接近1。因此，该公式会造成任何物品都会和热门物品有很大的相似度。为了避免推荐出热门的物品，可以用如下公式：

$$w_{i,j}=\frac{|N(i)\cap N(j)|}{\sqrt{|N(i)||N(j)|}}$$

这个公式惩罚了物品 j 的权重，因此减轻了热门物品与很多物品相似的可能性。

2. 欧氏距离（Euclidean Distance）

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

$$d(x,y)=\sqrt{\sum(x_i-y_i)^2}$$

可以看出，当 n=2 时，欧几里得距离就是平面上两个点的距离。

当用欧几里得距离表示相似度时，一般采用以下公式进行转换：距离越小，相似度越大。

$$\mathrm{sim}(x,y)=\frac{1}{1+d(x,y)}$$

3. 皮尔逊相关系数（Pearson Correlation Coefficient）
4. Cosine 相似度（Cosine Similarity）
5. Tanimoto 系数（Tanimoto Coefficient）

推荐计算

1. 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的邻居用户，然后将邻居用户喜欢的推荐给当前用户。在计算上，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及其对物品的偏好，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14-1 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。

用户/物品	物品A	物品B	物品C	物品D
用户A	√		√	推荐
用户B		√		
用户C	√		√	√

2. 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。

用户/物品	物品A	物品B	物品C
用户A	√		√
用户B	√	√	√
用户C	√		推荐

图 14-2　基于物品的 CF 的基本原理

根据用户评分矩阵采用同现相似度计算物品相似度矩阵。

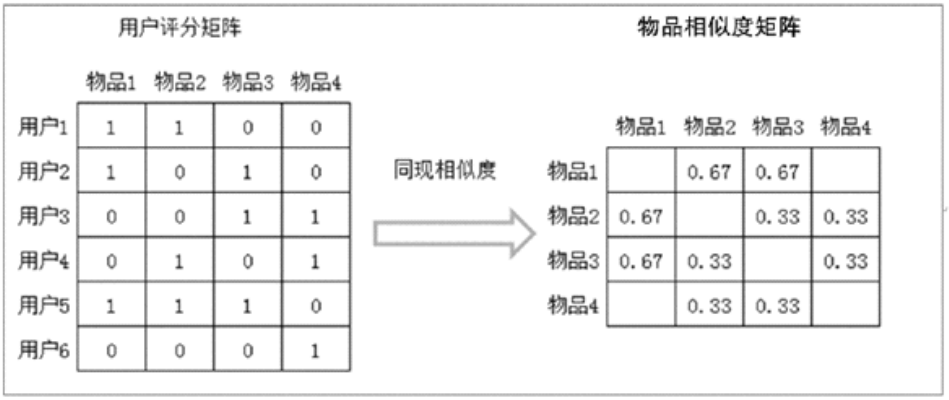


图 14-3　物品相似度矩阵

其相似度计算实现了分布式计算，实现过程如下：

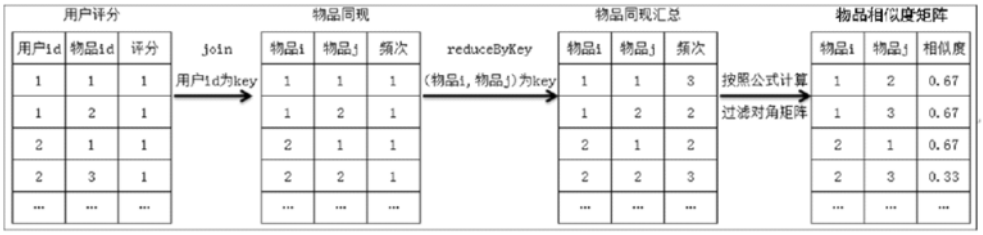


图 14-4　分布式同现相似度矩阵计算过程

对于欧氏相似度的计算，采用离散计算公式d(x, y) = sqrt(Σ((x(i)-y(i)) * (x(i)- y(i))))。其中，i 只取x、y 同现的点，未同现的点不参与相似度计算；sim(x, y) = m / (1 + d(x, y))，m 为x、y 重叠数，同现次数

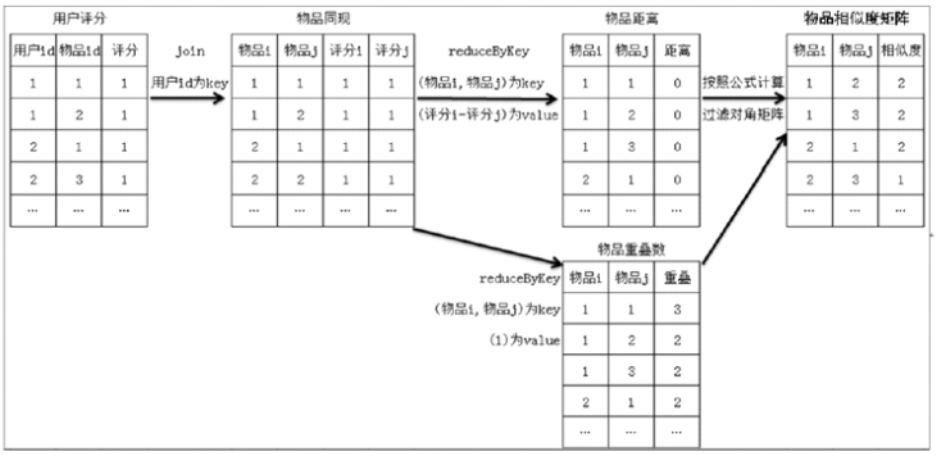


图 14-5　分布式欧氏距离相似度矩阵计算过程

根据物品相似度矩阵和用户评分计算用户推荐列表，计算公式是R=W*A，取推荐计算中用户未评分过的物品，并且按照计算结果倒序推荐给用户。

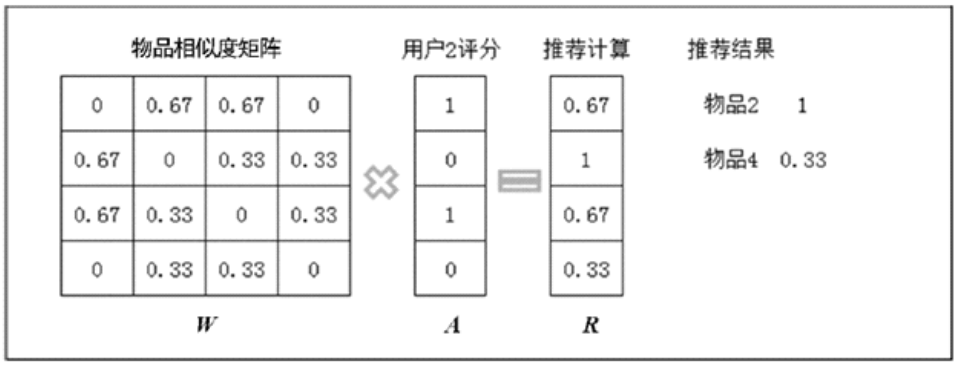


图 14-6　协同推荐计算

其推荐计算实现了分布式计算。

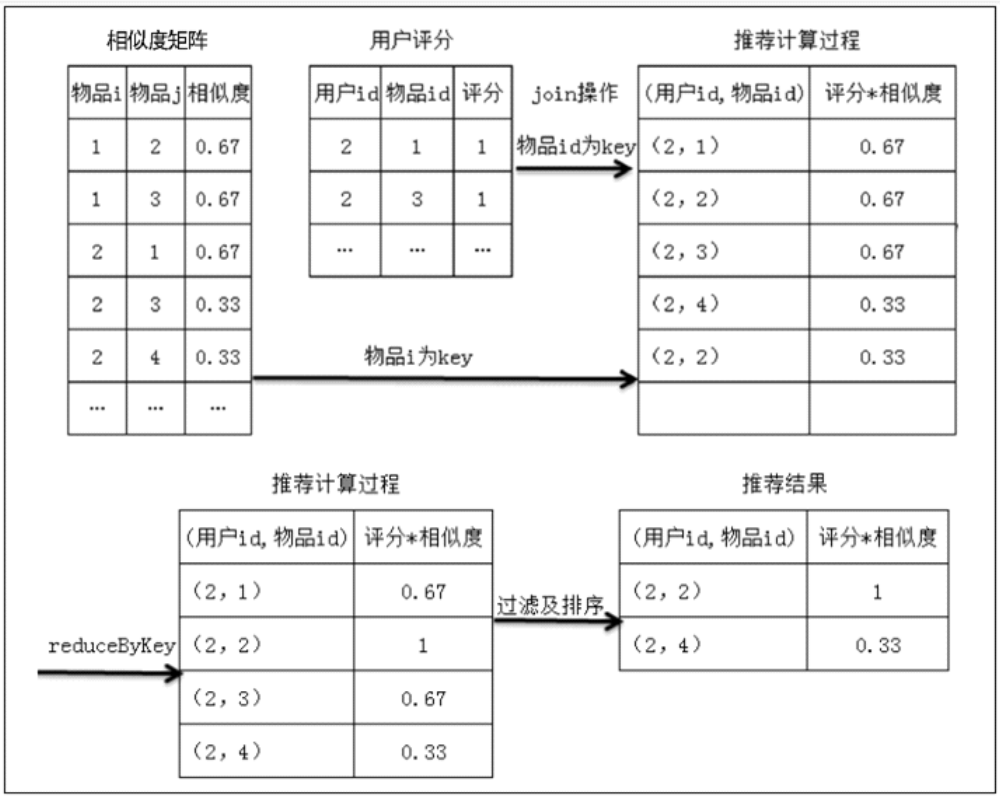


图 14-7　分布式协同推荐计算过程

```
 def main(args: Array[String]): Unit = {
  // 创建sc
  val conf = new SparkConf();
  conf.setAppName("SparkMlibDemo01")
  conf.setMaster("local")
  val sc = new SparkContext(conf);

  //导入开发包
  import org.apache.spark.mllib.recommendation.ALS
  import org.apache.spark.mllib.recommendation.MatrixFactorizationModel
  import org.apache.spark.mllib.recommendation.Rating

  // 加载并解析数据
  val data = sc.textFile("test.data")
  val ratings = data.map(_.split(',') match {
   case Array(user, item, rate) =>Rating(user.toInt, item.toInt, rate.toDouble)
  })

  // 使用ALS训练数据建立推荐模型
  val rank = 10
  val numIterations = 10
  val model = ALS.train(ratings, rank, numIterations, 0.01)

  // 从rating中获取user-product数据集
  val usersProducts = ratings.map {
   case Rating(user, product, rate) =>(user, product)
  }

  // 使用推荐模型预对用户和商品进行评分，得到预测评分的数据集
  val predictions =
   model.predict(usersProducts).map {
    case Rating(user, product, rate) =>((user, product), rate)
   }

  // 真实数据和预测数据进行合并
  val ratesAndPreds = ratings.map {
   case Rating(user, product, rate) =>((user, product), rate)
  }.join(predictions)

  val MSE = ratesAndPreds.map {
   case ((user, product), (r1, r2)) =>val err = (r1 - r2)
   err * err
  }.mean()

  // 计算MSE值，这个值越小，说明该model越接近正确值
  println("Mean Squared Error = " + MSE)

  // 存储、加载模型
//    model.save(sc, "myAModel")
//    val sameModel = MatrixFactorizationModel.load(sc, "myAModel")

  // 为指定用户推荐商品
  val userId = 1
  val K = 10
  val topKRecs = model.recommendProducts(userId, K)
  println(topKRecs.mkString("\n"))

 }
```

Spark实现推荐案例02

测试环境

为了测试简单，在本地以local方式运行Spark，你需要做的是下载编译好的压缩包解压即可，可以参考Spark本地模式运行。

测试数据使用MovieLens的MovieLens 10M数据集，下载之后解压到data目录。数据的格式请参考README中的说明，需要注意的是ratings.dat中的数据被处理过，每个用户至少访问了20个商品。

下面的代码均在spark-shell中运行，启动时候可以根据你的机器内存设置JVM参数，例如：

```
bin/spark-shell --executor-memory 3g --driver-memory 3g --driver-java-options '-Xms2g -Xmx2g -XX:+UseCompressedOops'
```

预测评分

这个例子主要演示如何训练数据、评分并计算根均方差。

准备工作

首先，启动spark-shell，然后引入mllib包，我们需要用到ALS算法类和Rating评分类：

```
import org.apache.spark.mllib.recommendation.{ALS, Rating}
```

Spark的日志级别默认为INFO，你可以手动设置为WARN级别，同样先引入log4j依赖：

```
import org.apache.log4j.{Logger,Level}
```

然后，运行下面代码：

```
Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)
```

加载数据

spark-shell启动成功之后，sc为内置变量，你可以通过它来加载测试数据：

```
val data = sc.textFile("data/ml-1m/ratings.dat")
```

接下来解析文件内容，获得用户对商品的评分记录：

```
val ratings = data.map(_.split("::") match { case Array(user, item, rate, ts) =>
  Rating(user.toInt, item.toInt, rate.toDouble)
}).cache()
```

查看第一条记录：

```
scala> ratings.first
res81: org.apache.spark.mllib.recommendation.Rating = Rating(1,1193,5.0)
```

我们可以统计文件中用户和商品数量：

```
val users = ratings.map(_.user).distinct()
val products = ratings.map(_.product).distinct()
println("Got "+ratings.count()+" ratings from "+users.count+" users on "+products.count+" products.")
```

可以看到如下输出：

```
//Got 1000209 ratings from 6040 users on 3706 products.
```

你可以对评分数据生成训练集和测试集，例如：训练集和测试集比例为8比2：

```
val splits = ratings.randomSplit(Array(0.8, 0.2), seed = 1111)
val training = splits(0).repartition(numPartitions)
val test = splits(1).repartition(numPartitions)
```

这里，我们是将评分数据全部当做训练集，并且也为测试集。

训练模型

接下来调用ALS.train()方法，进行模型训练：

```
val rank = 12
val lambda = 0.01
val numIterations = 20
val model = ALS.train(ratings, rank, numIterations, lambda)
```

训练完后，我们看看model中的用户和商品特征向量：

```
model.userFeatures
//res82: org.apache.spark.rdd.RDD[(Int, Array[Double])] = users MapPartitionsRDD[400] at mapValues at ALS.scala:218
model.userFeatures.count
//res84: Long = 6040

model.productFeatures
//res85: org.apache.spark.rdd.RDD[(Int, Array[Double])] = products MapPartitionsRDD[401] at mapValues at ALS.scala:222

model.productFeatures.count
//res86: Long = 3706
```

评测

我们要对比一下预测的结果，注意：**我们将训练集当作测试集**来进行对比测试。从训练集中获取用户和商品的映射：

```
val usersProducts= ratings.map { case Rating(user, product, rate) =>
  (user, product)
}
```

显然，测试集的记录数等于评分总记录数，验证一下：

```
usersProducts.count  //Long = 1000209
```

使用推荐模型对用户商品进行预测评分，得到预测评分的数据集：

```
var predictions = model.predict(usersProducts).map { case Rating(user, product, rate) =>
    ((user, product), rate)
}
```

查看其记录数：

```
predictions.count //Long = 1000209
```

将真实评分数据集与预测评分数据集进行合并，这样得到用户对每一个商品的实际评分和预测评分：

```
val ratesAndPreds = ratings.map { case Rating(user, product, rate) =>
  ((user, product), rate)
}.join(predictions)
ratesAndPreds.count  //Long = 1000209
```

然后计算根均方差：

```
val rmse= math.sqrt(ratesAndPreds.map { case ((user, product), (r1, r2)) =>
```

```
  val err = (r1 - r2)
  err * err
}.mean())
println(s"RMSE = $rmse")
```

上面这段代码其实就是对测试集进行评分预测并计算相似度，这段代码可以抽象为一个方法，如下：

```
/** Compute RMSE (Root Mean Squared Error). */
def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating]) = {
  val usersProducts = data.map { case Rating(user, product, rate) =>
    (user, product)
  }
val predictions = model.predict(usersProducts).map { case Rating(user, product, rate) =>
    ((user, product), rate)
  }

val ratesAndPreds = data.map { case Rating(user, product, rate) =>
    ((user, product), rate)
  }.join(predictions)

math.sqrt(ratesAndPreds.map { case ((user, product), (r1, r2)) =>
    val err = (r1 - r2)
    err * err
  }.mean())
}
```

除了RMSE指标，我们还可以及时AUC以及Mean average precision at K (MAPK)，关于AUC的计算方法，参考RunRecommender.scala，关于MAPK的计算方法可以参考《Packt.Machine Learning with Spark.2015.pdf》一书第四章节内容，或者你可以看本文后面内容。

保存真实评分和预测评分

我们还可以保存用户对商品的真实评分和预测评分记录到本地文件：

```
ratesAndPreds.sortByKey().repartition(1).sortBy(_._1).map({
  case ((user, product), (rate, pred)) => (user + "," + product + "," + rate + "," + pred)
}).saveAsTextFile("/tmp/result")
```

上面这段代码先按用户排序，然后重新分区确保目标目录中只生成一个文件。如果你重复运行这段代码，则需要先删除目标路径：

```
import scala.sys.process._
"rm -r /tmp/result".!
```

我们还可以对预测的评分结果按用户进行分组并按评分倒排序：

```
predictions.map { case ((user, product), rate) =>
  (user, (product, rate))
}.groupByKey(numPartitions).map{case (user_id,list)=>
  (user_id,list.toList.sortBy {case (goods_id,rate)=> - rate})
}
```

给一个用户推荐商品

这个例子主要是记录如何给一个或大量用户进行推荐商品，例如，对用户编号为384的用户进行推荐，查出该用户在测试集中评分过的商品。

找出5个用户：

```
users.take(5)
//Array[Int] = Array(384, 1084, 4904, 3702, 5618)
```

查看用户编号为384的用户的预测结果中预测评分排前10的商品：

```
val userId = users.take(1)(0) //384
val K = 10
val topKRecs = model.recommendProducts(userId, K)
println(topKRecs.mkString("\n"))
//     Rating(384,2545,8.354966018818265)
//     Rating(384,129,8.113083736094676)
//     Rating(384,184,8.038113395650853)
//     Rating(384,811,7.983433591425284)
//     Rating(384,1421,7.912044967873945)
//     Rating(384,1313,7.719639594879865)
//     Rating(384,2892,7.53667094600392)
//     Rating(384,2483,7.295378004543803)
//     Rating(384,397,7.141158013610967)
//     Rating(384,97,7.071089782695754)
```

查看该用户的评分记录：

```
val goodsForUser=ratings.keyBy(_.user).lookup(384)
// Seq[org.apache.spark.mllib.recommendation.Rating] = WrappedArray(Rating(384,2055,2.0), Rating(384,1197,4.0), Rating(384,593,5.0), Rating(384,599,3.0), Rating(384,673,2.0),
Rating(384,3037,4.0), Rating(384,1381,2.0), Rating(384,1610,4.0), Rating(384,3074,4.0), Rating(384,204,4.0), Rating(384,3508,3.0), Rating(384,1007,3.0), Rating(384,260,4.0),
Rating(384,3487,3.0), Rating(384,3494,3.0), Rating(384,1201,5.0), Rating(384,3671,5.0), Rating(384,1207,4.0), Rating(384,2947,4.0), Rating(384,2951,4.0), Rating(384,2896,2.0),
Rating(384,1304,5.0))
productsForUser.size //Int = 22
productsForUser.sortBy(-_.rating).take(10).map(rating => (rating.product, rating.rating)).foreach(println)
//     (593,5.0)
//     (1201,5.0)
//     (3671,5.0)
//     (1304,5.0)
//     (1197,4.0)
//     (3037,4.0)
//     (1610,4.0)
//     (3074,4.0)
//     (204,4.0)
//     (260,4.0)
```

可以看到该用户对22个商品评过分以及浏览的商品是哪些。

我们可以该用户对某一个商品的实际评分和预测评分方差为多少：

```
val actualRating = productsForUser.take(1)(0)
//actualRating: org.apache.spark.mllib.recommendation.Rating = Rating(384,2055,2.0)    val predictedRating = model.predict(789, actualRating.product)
val predictedRating = model.predict(384, actualRating.product)
//predictedRating: Double = 1.9426030777174637
val squaredError = math.pow(predictedRating - actualRating.rating, 2.0)
//squaredError: Double = 0.0032944066875075172
```

如何找出和一个已知商品最相似的商品呢？这里，我们可以使用余弦相似度来计算：

```
import org.jblas.DoubleMatrix
/* Compute the cosine similarity between two vectors */
def cosineSimilarity(vec1: DoubleMatrix, vec2: DoubleMatrix): Double = {
```

```
  vec1.dot(vec2) / (vec1.norm2() * vec2.norm2())
}
```

以2055商品为例，计算实际评分和预测评分相似度

```
val itemId = 2055
val itemFactor = model.productFeatures.lookup(itemId).head
//itemFactor: Array[Double] = Array(0.3660752773284912, 0.43573060631752014, -0.3421429991722107, 0.44382765889167786, -1.4875195026397705,
0.6274569630622864, -0.3264533579349518, -0.9939845204353333, -0.8710321187973022, -0.7578890323638916, -0.14621856808662415, -0.7254264950752258)
val itemVector = new DoubleMatrix(itemFactor)
//itemVector: org.jblas.DoubleMatrix = [0.366075; 0.435731; -0.342143; 0.443828; -1.487520; 0.627457; -0.326453; -0.993985; -0.871032; -0.757889; -0.146219; -0.725426]
cosineSimilarity(itemVector, itemVector)
// res99: Double = 0.9999999999999999
```

找到和该商品最相似的10个商品：

```
val sims = model.productFeatures.map{ case (id, factor) =>
  val factorVector = new DoubleMatrix(factor)
  val sim = cosineSimilarity(factorVector, itemVector)
  (id, sim)
}
val sortedSims = sims.top(K)(Ordering.by[(Int, Double), Double] { case (id, similarity) => similarity })
//sortedSims: Array[(Int, Double)] = Array((2055,0.9999999999999999), (2051,0.9138311231145874), (3520,0.8739823400539756), (2190,0.8718466671129721), (2050,0.8612639515847019),
(1011,0.8466911667526461), (2903,0.8455764332511272), (3121,0.8227325520485377), (3674,0.8075743004357392), (2016,0.8063817280259447))
println(sortedSims.mkString("\n"))
//     (2055,0.9999999999999999)
//     (2051,0.9138311231145874)
//     (3520,0.8739823400539756)
//     (2190,0.8718466671129721)
//     (2050,0.8612639515847019)
//     (1011,0.8466911667526461)
//     (2903,0.8455764332511272)
//     (3121,0.8227325520485377)
//     (3674,0.8075743004357392)
//     (2016,0.8063817280259447)
```

显然第一个最相似的商品即为该商品本身，即2055，我们可以修改下代码，取前k+1个商品，然后排除第一个：

```
val sortedSims2 = sims.top(K + 1)(Ordering.by[(Int, Double), Double] { case (id, similarity) => similarity })
//sortedSims2: Array[(Int, Double)] = Array((2055,0.9999999999999999), (2051,0.9138311231145874), (3520,0.8739823400539756), (2190,0.8718466671129721), (2050,0.8612639515847019),
(1011,0.8466911667526461), (2903,0.8455764332511272), (3121,0.8227325520485377), (3674,0.8075743004357392), (2016,0.8063817280259447), (3672,0.8016276723120674))
sortedSims2.slice(1, 11).map{ case (id, sim) => (id, sim) }.mkString("\n")
//     (2051,0.9138311231145874)
//     (3520,0.8739823400539756)
//     (2190,0.8718466671129721)
//     (2050,0.8612639515847019)
//     (1011,0.8466911667526461)
//     (2903,0.8455764332511272)
//     (3121,0.8227325520485377)
//     (3674,0.8075743004357392)
//     (2016,0.8063817280259447)
//     (3672,0.8016276723120674)
```

接下来，我们可以计算给该用户推荐的前K个商品的平均准确度MAPK，该算法定义如下（该算法是否正确还有待考证）：

```
/* Function to compute average precision given a set of actual and predicted ratings */
// Code for this function is based on: https://github.com/benhamner/Metrics
def avgPrecisionK(actual: Seq[Int], predicted: Seq[Int], k: Int): Double = {
  val predK = predicted.take(k)
  var score = 0.0
  var numHits = 0.0
  for ((p, i) <- predK.zipWithIndex) {
    if (actual.contains(p)) {
      numHits += 1.0
      score += numHits / (i.toDouble + 1.0)
    }
  }
  if (actual.isEmpty) {
    1.0
  } else {
    score / scala.math.min(actual.size, k).toDouble
  }
}
```

给该用户推荐的商品为：

```
val actualProducts = productsForUser.map(_.product)
//actualProducts: Seq[Int] = ArrayBuffer(2055, 1197, 593, 599, 673, 3037, 1381, 1610, 3074, 204, 3508, 1007, 260, 3487, 3494, 1201, 3671, 1207, 2947, 2951, 2896, 1304)
```

给该用户预测的商品为：

```
 val predictedProducts = topKRecs.map(_.product)
//predictedProducts: Array[Int] = Array(2545, 129, 184, 811, 1421, 1313, 2892, 2483, 397, 97)
```

最后的准确度为：

```
val apk10 = avgPrecisionK(actualProducts, predictedProducts, 10)
// apk10: Double = 0.0
```

批量推荐

你可以评分记录中获得所有用户然后依次给每个用户推荐：

```
val users = ratings.map(_.user).distinct()
users.collect.flatMap { user =>
  model.recommendProducts(user, 10)
}
```

这种方式是遍历内存中的一个集合然后循环调用RDD的操作，运行会比较慢，另外一种方式是直接操作model中的userFeatures和productFeatures，代码如下：

```
val itemFactors = model.productFeatures.map { case (id, factor) => factor }.collect()
val itemMatrix = new DoubleMatrix(itemFactors)
println(itemMatrix.rows, itemMatrix.columns)
//(3706,12)
// broadcast the item factor matrix
val imBroadcast = sc.broadcast(itemMatrix)

//获取商品和索引的映射
var idxProducts=model.productFeatures.map { case (prodcut, factor) => prodcut }.zipWithIndex().map{case (prodcut, idx) => (idx,prodcut)}.collectAsMap()
val idxProductsBroadcast = sc.broadcast(idxProducts)
```

```
val allRecs = model.userFeatures.map{ case (user, array) =>
  val userVector = new DoubleMatrix(array)
  val scores = imBroadcast.value.mmul(userVector)
  val sortedWithId = scores.data.zipWithIndex.sortBy(-_._1)
  //根据索引取对应的商品id
  val recommendedProducts = sortedWithId.map(_._2).map{idx=>idxProductsBroadcast.value.get(idx).get}
  (user, recommendedProducts)
}
```

这种方式其实还不是最优方法，更好的方法可以参考[Personalised recommendations using Spark]，当然这篇文章中的代码还可以继续优化一下。我修改后的代码如下，供大家参考：

```
val productFeatures = model.productFeatures.collect()
var productArray = ArrayBuffer[Int]()
var productFeaturesArray = ArrayBuffer[Array[Double]]()
for ((product, features) <- productFeatures) {
  productArray += product
  productFeaturesArray += features
}
val productArrayBroadcast = sc.broadcast(productArray)
val productFeatureMatrixBroadcast = sc.broadcast(new DoubleMatrix(productFeaturesArray.toArray).transpose())

start = System.currentTimeMillis()
val allRecs = model.userFeatures.mapPartitions { iter =>
  // Build user feature matrix for jblas
  var userFeaturesArray = ArrayBuffer[Array[Double]]()
  var userArray = new ArrayBuffer[Int]()
  while (iter.hasNext) {
    val (user, features) = iter.next()
    userArray += user
    userFeaturesArray += features
  }

var userFeatureMatrix = new DoubleMatrix(userFeaturesArray.toArray)
  var userRecommendationMatrix = userFeatureMatrix.mmul(productFeatureMatrixBroadcast.value)
  var productArray=productArrayBroadcast.value
  var mappedUserRecommendationArray = new ArrayBuffer[String](params.topk)

// Extract ratings from the matrix
  for (i <- 0 until userArray.length) {
    var ratingSet =  mutable.TreeSet.empty(Ordering.fromLessThan[(Int,Double)](_._2 > _._2))
    for (j <- 0 until productArray.length) {
      var rating = (productArray(j), userRecommendationMatrix.get(i,j))
      ratingSet += rating
    }
    mappedUserRecommendationArray += userArray(i)+","+ratingSet.take(params.topk).mkString(",")
  }
  mappedUserRecommendationArray.iterator
}
```

2015.06.12 更新：

悲哀的是，上面的方法还是不能解决问题，因为矩阵相乘会撑爆集群内存；可喜的是，如果你关注Spark最新动态，你会发现Spark1.4.0中MatrixFactorizationModel提供了recommendForAll方法实现离线批量推荐，详细说明见[SPARK-3066]。因为，我使用的Hadoop版本是CDH-5.4.0，其中Spark版本还是1.3.0，所以暂且不能在集群上测试Spark1.4.0中添加的新方法。

如果上面结果跑出来了，就可以验证推荐结果是否正确。还是以384用户为例：

```
allRecs.lookup(384).head.take(10)
//res50: Array[Int] = Array(1539, 219, 1520, 775, 3161, 2711, 2503, 771, 853, 759)
topKRecs.map(_.product)
//res49: Array[Int] = Array(1539, 219, 1520, 775, 3161, 2711, 2503, 771, 853, 759)
```

接下来，我们可以计算所有推荐结果的准确度了，首先，得到每个用户评分过的所有商品：

```
val userProducts = ratings.map{ case Rating(user, product, rating) => (user, product) }.groupBy(_._1)
```

然后，预测的商品和实际商品关联求准确度：

```
// Finally, compute the APK for each user, and average them to find MAPK
val MAPK = allRecs.join(userProducts).map{ case (userId, (predicted, actualWithIds)) =>
  val actual = actualWithIds.map(_._2).toSeq
  avgPrecisionK(actual, predicted, K)
}.reduce(_ + _) / allRecs.count
println("Mean Average Precision at K = " + MAPK)
//Mean Average Precision at K = 0.018827551771260383
```

其实，我们也可以使用Spark内置的算法计算RMSE和MAE：

```
// MSE, RMSE and MAE
import org.apache.spark.mllib.evaluation.RegressionMetrics
val predictedAndTrue = ratesAndPreds.map { case ((user, product), (actual, predicted)) => (actual, predicted) }
val regressionMetrics = new RegressionMetrics(predictedAndTrue)
println("Mean Squared Error = " + regressionMetrics.meanSquaredError)
println("Root Mean Squared Error = " + regressionMetrics.rootMeanSquaredError)
// Mean Squared Error = 0.5490153087908566
// Root Mean Squared Error = 0.7409556726220918

// MAPK
import org.apache.spark.mllib.evaluation.RankingMetrics
val predictedAndTrueForRanking = allRecs.join(userProducts).map{ case (userId, (predicted, actualWithIds)) =>
  val actual = actualWithIds.map(_._2)
  (predicted.toArray, actual.toArray)
}
val rankingMetrics = new RankingMetrics(predictedAndTrueForRanking)
println("Mean Average Precision = " + rankingMetrics.meanAveragePrecision)
// Mean Average Precision = 0.04417535679520426
```

计算推荐2000个商品时的准确度为：

```
val MAPK2000 = allRecs.join(userProducts).map{ case (userId, (predicted, actualWithIds)) =>
  val actual = actualWithIds.map(_._2).toSeq
  avgPrecisionK(actual, predicted, 2000)
}.reduce(_ + _) / allRecs.count
println("Mean Average Precision = " + MAPK2000)
//Mean Average Precision = 0.025228311843069083
```

保存和加载推荐模型

对与实时推荐，我们需要启动一个web server，在启动的时候生成或加载训练模型，然后提供API接口返回推荐接口，需要调用的相关方法为：

```
save(model: MatrixFactorizationModel, path: String)
```

```
load(sc: SparkContext, path: String)
```

model中的userFeatures和productFeatures也可以保存起来：

```
val outputDir="/tmp"
model.userFeatures.map{ case (id, vec) => id + "\t" + vec.mkString(",") }.saveAsTextFile(outputDir + "/userFeatures")
model.productFeatures.map{ case (id, vec) => id + "\t" + vec.mkString(",") }.saveAsTextFile(outputDir + "/productFeatures")
```

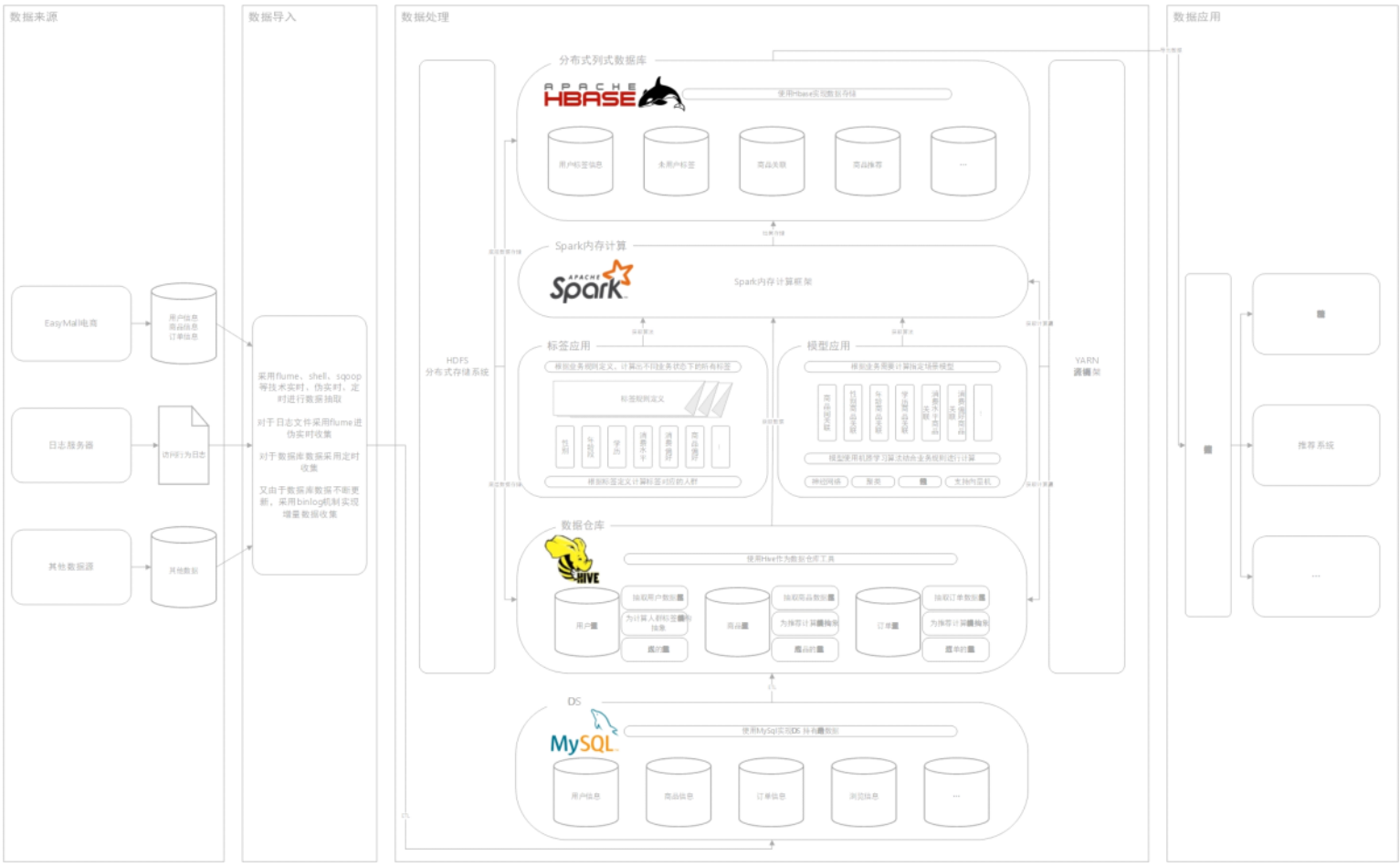
总结

本文主要记录如何使用ALS算法实现协同过滤并给用户推荐商品，以上代码在Github仓库中的ScalaLocalALS.scala文件。

如果你想更加深入了解Spark MLlib算法的使用，可以看看Packt.Machine Learning with Spark.2015.pdf这本电子书并下载书中的源码，本文大部分代码参考自该电子书

来自 <http://blog.javachen.com/2015/06/01/how-to-implement-collaborative-filtering-using-spark-als.html>

推荐系统架构设计


数据来源
数据导入
数据处理
数据应用
EasyMall电商
用户信息
商品信息
订单信息
日志服务器
访问行为日志
其他数据源
其他数据
采用flume、shell、sqoop等技术实时、伪实时、定时进行数据抽取
对于日志文件采用flume进伪实时收集
对于数据库数据采用定时收集
又由于数据库数据不断更新，采用binlog机制实现增量数据收集
分布式列式数据库
HBASE
使用Hbase实现数据存储
用户标签信息
未用户标签
商品关联
商品推荐
...
Spark内存计算
Spark内存计算框架
HDFS
分布式存储系统
标签应用
根据业务规则定义，计算出不同业务状态下的所有标签
标签规则定义
根据标签定义计算标签对应的人群
模型应用
根据业务需要计算指定场景模型
模型使用机器学习算法结合业务规则进行计算
神经网络
聚类
YARN
资源调度框架
数据仓库
HIVE
使用Hive作为数据仓库工具
用户层
抽取用户数据层
为计算人群标签做准备的数据
商品层
抽取商品数据层
为推荐计算做准备的数据
订单层
抽取订单数据层
为推荐计算做准备的数据
DS
MySQL
使用MySql实现DS 持有最新数据
用户信息
商品信息
订单信息
浏览信息
...
推荐系统
...