协同过滤（Collective Filtering）可以说是推荐系统的标配算法。

在谈推荐必谈协同的今天，我们也来谈一谈基于KNN的协同过滤在实际的推荐应用中的一些心得体会。

我们首先从协同过滤的两个假设聊起。

两个假设：

1. 用户一般会喜欢与自己喜欢物品相似的物品
2. 用户一般会喜欢与自己相似的其他用户喜欢的物品

上述假设分别对应了协同过滤的两种实现方式：基于物品相似（item_cf）及基于用户相似（user_cf）。

因此，协同过滤在实现过程中，最本质的任务就是计算相似度的问题，包括计算item之间的相似度，或user之间的相似度。要计算相似度，就需要一个用来计算相似度的表达。

输入数据：

而协同过滤的输入是一个用户对item的评分矩阵，如下图所示：

items \ users	1	2	3	4	5	6	7	8	9	10	11	12
1	1		3		?	5			5		4	
2			5	4			4			2	1	3
3	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2					2	5
6	1		3		3			2			4	

这种输入数据中，没有关于item或user的任何扩展描述。

因此item和user之间只能互为表达，用所有用户对某item的评分来描述一个item，以及用某个用户所有评分的item来描述该用户本身。分别对应途中的行向量和列向量。

这样，我们只需要搞定向量之间相似度的计算问题就OK了。

相似度计算：

以Item相似度为例，我们一般使用如下的公式完成相似度的计算：

$$w_{ij} = \frac{(\sum_{u \in U(i,j)} r'_{ui} * r'_{uj}) * (|U(i,j)| - 1)}{\sqrt{\sum_{u \in U(i,j)} r'^{2}_{ui} * \sum_{u \in U(i,j)} r'^{2}_{uj}} * (|U(i,j)| - 1 + \lambda)}$$

其中：

Wij表示标号为i，j的两个item的相似度

U(i,j)表示同时对i，j有评分的用户的集合

Rui表示用户u对item i的评分

参数lambda为平滑（惩罚）参数

当然，相似度的计算有很多可选的方案， 例如直接计算两个向量夹角的cos值。只是通过实验发现，文中提到的方法在Lz的实际推荐应用中，效果要略好而已。

对于user相似度则只需要把输入矩阵转置看待即可。

评分预测过程：

仍以item_cf为例，在得到item相似度之后，可以通过一个矩阵乘法运算来为输入数据中的空白处计算一个预测值。

将输入矩阵看成一个n*m的矩阵，记为A，n为用户数，m为item数。而计算得到的相似度矩阵为一个m*m的矩阵，记为B。

则预测评分的过程就是A*B的过程，结果也是一个n*m的矩阵。

具体公式如下：

$$r_{ui}^{predict} = b_i + \sum_{j \in S^k(i;u)} r'_{uj} w_{ji}$$

其中bi为歌曲本身的流行程度，可通过其他方式计算得到。

相似度计算的复杂度：

item相似度计算的复杂度：输入矩阵A(n*m) 中的用户数为n，item数为m，则要计算出一个m*m的相似度矩阵的话共需要O=m*m*n*d，d为数据的稀疏度。

假设n=200w，m=5w，数据稀疏度为1%，则O=50万亿。好在该过程可以很方便地使用map-reducer的方式在

hdfs上分布式实现。

再假设你通过1000个计算节点来完成，则每个计算节点上的复杂度O1=5000亿。

user相似度计算的复杂度：O=n*n*m*d，假设计算资源仍为1000，则每个计算节点的复杂度为O1=2000万亿。

相似度计算的两种实现方式：

一般来讲，item的量都是比较小的，而且固定，而用户的量会比较大，而且每天都可能会不一样。

因此，在计算过程中的实用策略也不同，大体上有如下两种：

1. 倒排式计算：

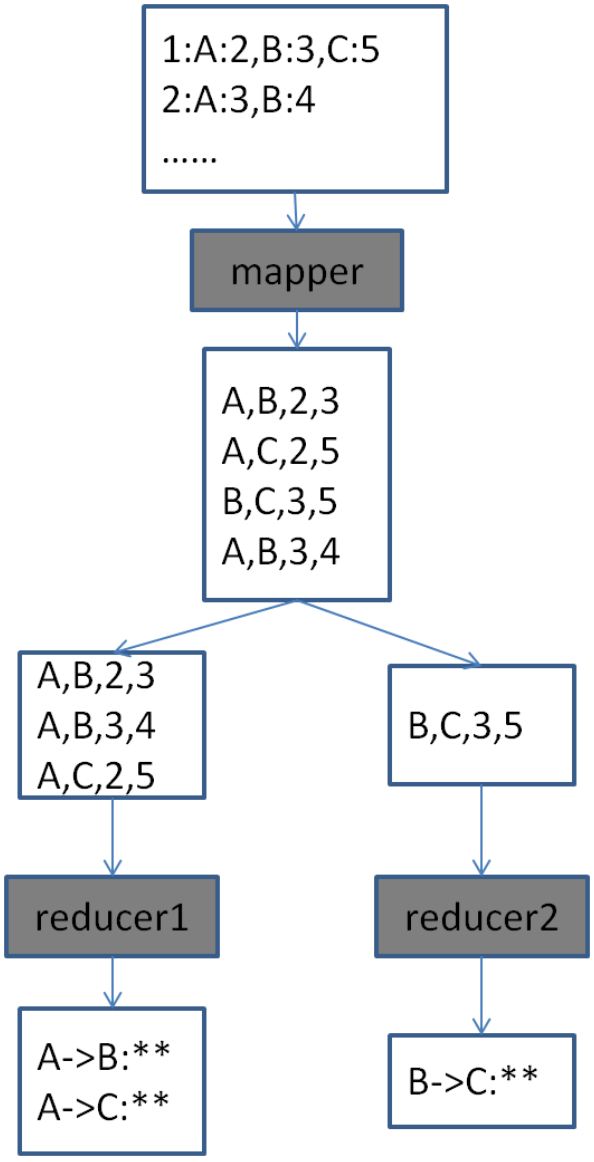
这种方式尤其适用于用户量较大，而item量较小，且每个用户已评分的item数较少的情况。

具体做法就是：

在MR的map阶段将每个用户的评分item组合成pair <left,right,leftscore,rightscore> 输出，left作为分发键，left+right作为排序键。

在reduce阶段，将map中过来的数据扫一遍即可求得所有item的相似度。

详细Hadoop过程如下所示：



该方式的好处在于没有关联的item之间不会产生相似度的计算开销，所谓没有关联指没有任何一个用户对这两个item都有评分。

不好的地方在于，如果某些用户评分数据较多，则产生的pair对会十分巨大，在map和reducer之间会有极大的IO开销。

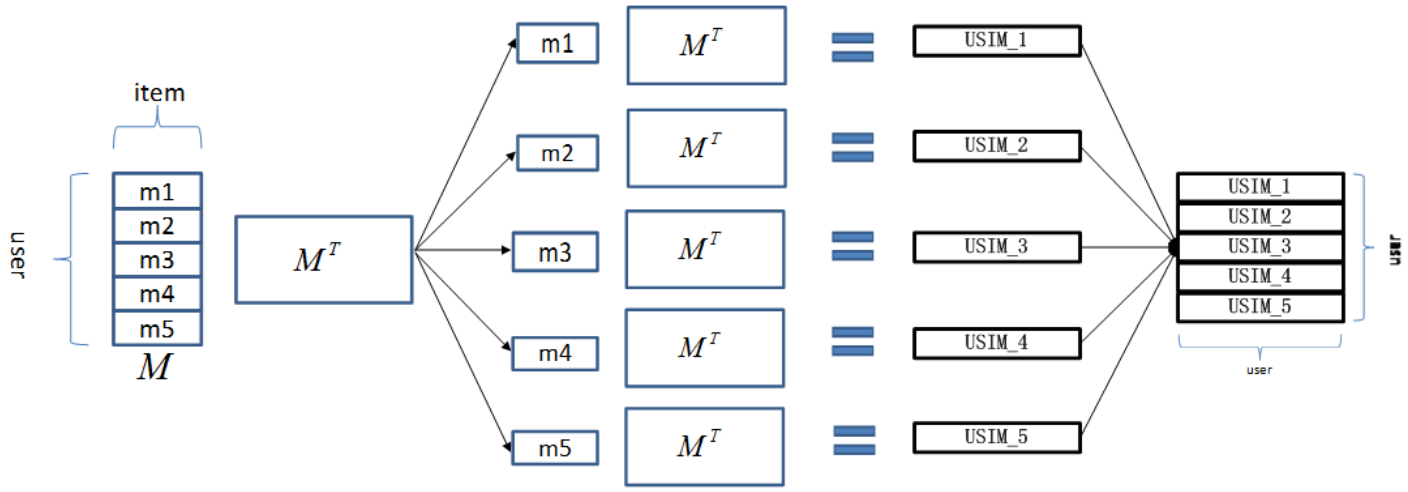
2. 矩阵分块计算：

这种计算方式适用于用户量远大于item量的情况下，用户相似度的计算过程。

具体做法：将评分矩阵M切分成多个小块，每个小块与原矩阵的转置矩阵做类似矩阵乘法的运算（按照文中提到的相似度计算公式，而非向量内积），

最后将计算得到的结果合并即可。

详细Hadoop过程如下所示：



该方式的好处是避免map与reducer之间的大量缓存，而且这种方式压根不需要reducer。

不好的地方在于transpose(M)需要在任务开始计算前缓存在每个hadoop计算节点，在矩阵M较大的时候会影响任务的启动效率。

两种方式各有优劣，精确计算user之间或item之间的相似度时需要根据实际的数据特点选择合适的计算方式。

然而，随着业务的增长，评分矩阵会越来越大，同时也会越来越稠密。

这时无论采用哪一种方式，精确地计算user之间或item之间的相似度已可望而不可及。

而且，考虑实际的推荐场景，在数据足够多的情况下，相似度的精确计算也没有必要。

基于**SimHash**的相似度计算：

当数据量太大时，往往只需要求得一个与最优解相近的近似解即可，相似度的计算也是如此。

基于SimHash计算用户之间或item之间的相似度是推荐中较为常用的技巧。

该方法之所以能够work，主要基于如下两点：1.hash的随机性，2.数据足够多。

这两点决定了通过simhash计算得到的相似度结果是否靠谱。

而关于SimHash的具体原理，本文不做详述，不熟悉的同学可参阅链接simhash算法原理或其他相关资料。

这里主要介绍下自己在实际工作中的做法：

首先**Hash**函数的选择：

本人在实际工作中尝试了两种Hash函数：

1. DJBX33A，将字符串映射成一个64位的无符号整数。具体实现参考：DJBX33A哈希函数实现

2. MD5， 将字符串映射成一个128为的二进制流。本人在实际工作中使用char[16]来表示。

汉明距离与相似度：

在计算得到每个对象的hashsign之后，可进一步得到两个hashsign的汉明距离。假设最终的sign为n维，则汉明距离最大为n，最小为0.分别对应相似度为-1，1。

当汉明距离为n/2时，两个hashsign对应的对象之间的相似度为0，也可以理解为他们的夹角为 PI/2。

可以通过如下公式对汉明距离与相似度之间做转换：

$$w_{ij} = \cos(PI * H_{ij} / N)$$

其中N为sign维数，Hij为i，j之间的汉明距离。

汉明距离的计算：

1. 如果hashcode在64位以内，最终计算的hashsign可以使用计算机内置的类型来表示，如unsigned long。

   然后对两个对象的hashsig按位异或，计算结果中有多少位为1，即为汉明距离值。

2. 如果hashcode大于64位，例如MD5，则需要一个复合结构来表示。我们可以使用char[16]来表示128bit。

   然后在计算汉明距离时分别使用两个unsigned long 来表示高位的64bit和低位的64bit。

实践证明，通过simhash计算的到的相似度，与真实的相似度之间几乎等同。

而且，在实际应用中，通过simhash的推荐结果，鲁棒性也更强。