**[Apache Mahout的Taste基于Hadoop实现协同过滤推荐引擎的代码分析](http://www.cnblogs.com/jerome-rong/archive/2012/05/22/2513179.html)**

Taste 是 Apache Mahout 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。  
该推荐引擎是用<userid,itemid,preference>这样简单的数据格式表达用户对物品的偏好。  
以此为输入数据，计算后就可以得到为每个user推荐的items列表。  
  
他提供了方便的单机版的编程接口，也提供了基于hadoop的分布式的实现。  
单机版的编程接口主要适用于写demo和做算法的评估，若处理大规模数据，还是需分布式的实现。  
以下是对org.apache.mahout.cf.taste.hadoop.item.RecommenderJob的各MapReduce步骤的一个解读。  
Taste 实现一个分布式的协同过滤推荐共经历了如下12个MapReduce步骤。  
以下分析了各步骤的Mapper和Reducer都做了哪些工作，并有什么格式的数据输出。  
  
**代码分析：**  
1、计算item的itemid\_index和最小itemid值  
1.1、ItemIDIndexMapper.class, VarIntWritable.class, VarLongWritable.class,  
用原始输入，将userid,itemid,pref数据转成itemid\_index,itemid   
1.2、ItemIDIndexReducer.class, VarIntWritable.class, VarLongWritable.class,  
在itemid\_index,Iterator<itemid>中找最小的itemid，输出itemid\_index,minimum\_itemid  
此处只是保存一个int型的itemid\_index索引和对应的long型的itemid的映射  
  
2、计算各user的item偏好向量，即Vector<item,pref>  
2.1、ToItemPrefsMapper.class, VarLongWritable.class, booleanData ? VarLongWritable.class : EntityPrefWritable.class,  
用原始输入，读入偏好数据，得到userid,<itemid,pref>  
2.2、ToUserVectorReducer.class, VarLongWritable.class, VectorWritable.class,  
将userid,Iterator<itemid,pref>中的itemid变成itemid\_index，得到userid,Vector<itemid\_index,pref>，后者用RandomAccessSparseVector来存。  
  
3、统计数据中有多少个user  
3.1、CountUsersMapper.class,CountUsersKeyWritable.class,VarLongWritable.class,  
用步骤2的输出，统计独立userid数目，先转换数据为userid,userid  
3.2、CountUsersReducer.class,VarIntWritable.class,NullWritable.class,  
通过CountUsersPartitioner将所有数据发到一个区，一个Reducer来处理  
由于userid都已排序，所以可以用极简单的方式来统计出独立userid数  
输出只有一个值，即用户数  
  
4、计算item的user偏好向量，即Vector<userid,pref>，也即拿步骤2的结果做矩阵的修剪和转置  
4.1、MaybePruneRowsMapper.class,IntWritable.class,DistributedRowMatrix.MatrixEntryWritable.class,  
用步骤2的输出，按指定的maxCooccurrences参数值来修剪Vector的数目，目的是控制计算的规模，减少计算量  
然后转为以userid\_index为列号、itemid\_index为行号、pref为值的矩阵，用MatrixEntryWritable表示矩阵。  
输出为itemid\_index,Matrix<userid\_index,itemid\_index,pref>  
4.2、ToItemVectorsReducer.class,IntWritable.class,VectorWritable.class,  
输出为itemid\_index,Vector<userid\_index,pref>，相当于对步骤2的结果进行了矩阵的转置，  
  
有了偏好矩阵数据，接下来会调用RowSimilarityJob来计算行的相似度  
此处的行是item，所以默认是item-base的CF。  
但其实可以通过传入是否转置的参数来对步骤1进行调整，将userid和itemid转换，就可以实现user-base的CF。  
此处也可以通过similarityClassname参数来指定用哪种算法来计算相似度。  
RowSimilarityJob将通过接下来的3个步骤来实现：  
  
5、用相似度算法给向量赋权  
5.1、RowWeightMapper.class,VarIntWritable.class,WeightedOccurrence.class,  
用相应的相似度算法来计算步骤4的输出，计算每个itemid\_index所对应的Vector<userid\_index,pref>的weight。  
输出为userid\_index,WeightedOccurrence<itemid\_index,pref,weight>，WeightedOccurrence是一个简单的数据封装类。  
5.2、WeightedOccurrencesPerColumnReducer.class,VarIntWritable.class,WeightedOccurrenceArray.class,  
将Iterator<WeightedOccurrence>简单变为WeightedOccurrenceArray，后者只是简单继承了ArrayWritable。  
最后输出结果为userid\_index,WeightedOccurrenceArray，数组的数据项是WeightedOccurrence<itemid\_index,pref,weight>  
  
6、用相似度算法计算相似度，得到相似度矩阵  
6.1、CooccurrencesMapper.class,WeightedRowPair.class,Cooccurrence.class,  
取出步骤5的结果，将WeightedOccurrenceArray的数据双重循环，拼装如下的KV数据结构  
WeightedRowPair<itemid\_indexA,itemid\_indexB,weightA,weightB>,Cooccurrence<userid\_index,prefA,prefB>  
6.2、SimilarityReducer.class,SimilarityMatrixEntryKey.class,DistributedRowMatrix.MatrixEntryWritable.class,  
此步骤的Map输出，也即Reduce的输入是WeightedRowPair<itemid\_indexA,itemid\_indexB,weightA,weightB>, Iterator<Cooccurrence<userid\_index,prefA,prefB>>  
也即itemA和itemB的weight，以及不同user对itemA和itemB的pref。  
相应的Similarity实例就可以利用以上数据计算itemA与itemB的相似度评分similarityValue  
输出结果为SimilarityMatrixEntryKey<itemid\_indexA,similarityValue>,Matrix<itemid\_indexA,itemid\_indexB,similarityValue>  
也就是不同item和itemA的俩俩相似度，得到一个相似度矩阵  
  
7、将相似度矩阵转为向量存储  
7.1、Mapper.class,SimilarityMatrixEntryKey.class,DistributedRowMatrix.MatrixEntryWritable.class,  
将步骤6的结果简单读入，item相似度矩阵  
7.2、EntriesToVectorsReducer.class,IntWritable.class,VectorWritable.class,  
输出为itemid\_indexA,Vector<itemid\_indexX,similarityValue>，Vector用SequentialAccessSparseVector存储。  
也就是输出为不同的其他item与itemA之间的相似度值  
  
8、PartialMultiply的预处理1，填充vector部分的数据  
8.1、SimilarityMatrixRowWrapperMapper.class, VarIntWritable.class, VectorOrPrefWritable.class,  
用步骤7的相似度数据，输出itemid\_index,VectorOrPrefWritable(vector,null,null)  
8.2、Reducer.class, VarIntWritable.class, VectorOrPrefWritable.class,  
默认Reducer，直接输出Mapper的输出  
  
9、PartialMultiply的预处理2，填充userid和pref部分的数据  
9.1、UserVectorSplitterMapper.class, VarIntWritable.class, VectorOrPrefWritable.class,  
如果提供了一个userid列表文件，Mapper初始化时会先读入该文件到FastIDSet<userid>中  
如果userid不在这个Set中，则会直接return，也就是只会为该列表中的user做推荐  
用步骤2的用户对各item的偏好数据，输出itemid\_index,VectorOrPrefWritable(null,userid,pref)  
9.2、Reducer.class, VarIntWritable.class, VectorOrPrefWritable.class,  
默认Reducer，直接输出Mapper的输出  
  
10、拼装两个PartialMultiply预处理的数据  
10.1、Mapper.class, VarIntWritable.class, VectorOrPrefWritable.class,  
用FileInputFormat.setInputPaths指定多个路径，将步骤8和9的输出同时作为输入  
10.2、ToVectorAndPrefReducer.class, VarIntWritable.class, VectorAndPrefsWritable.class,  
将VectorOrPrefWritable(vector,null,null)和VectorOrPrefWritable(null,userid,pref)  
变为VectorAndPrefsWritable(vector,List<userid>,List<pref>)  
最后的输出是itemid\_index,VectorAndPrefsWritable(vector,List<userid>,List<pref>)  
  
11、如果设置了item过滤文件则读取，作为黑名单  
11.1、ItemFilterMapper.class, VarLongWritable.class, VarLongWritable.class,  
简单读入item过滤文件，输出为itemid,userid，这相当于“黑”名单，用于后面推荐结果的过滤。  
11.2、ItemFilterAsVectorAndPrefsReducer.class, VarIntWritable.class, VectorAndPrefsWritable.class,  
输出为itemid\_index,VectorAndPrefsWritable(vector,List<userid>,List<pref>)  
其中vector的值为vector(itemid\_index,Double.NaN)，pref的值都用1.0f来填充。  
注意，vector的第二项数据，也即similarityValue被设置为Double.NaN，后面将会用这个来判断这是否是黑名单。  
  
12、用相似度矩阵的PartialMultiply做推荐计算  
12.1、PartialMultiplyMapper.class, VarLongWritable.class, PrefAndSimilarityColumnWritable.class,  
如果步骤11存在，则用FileInputFormat.setInputPaths指定多个路径，将步骤10和11的输出同时作为输入  
也即输入为itemid\_index,VectorAndPrefsWritable(vector,List<userid>,List<pref>)，其中vector的值为Vector<itemid\_index,similarityValue>  
输出为userid,PrefAndSimilarityColumnWritable(pref,vector<itemid\_index,similarityValue>)  
12.2、AggregateAndRecommendReducer.class, VarLongWritable.class, RecommendedItemsWritable.class,  
初始化时，会读入步骤1的结果，是一个HashMap<itemid\_index,itemid>，也即index和itemid的映射  
若设置了item白名单文件，则初始化时也会读入文件到FastIDSet<itemid>，推荐结果必须在这里边。和步骤11的黑名单相反。  
Reducer在处理时会区分是否是booleanData而用不同的处理逻辑，此处我们主要讨论非booleanData，也即有实际pref数据的情况而不是默认用1.0f来填充的pref。  
Reducer中进行PartialMultiply，按乘积得到的推荐度的大小取出最大的几个item。  
处理的过程中需要将itemid\_index通过HashMap转换回itemid，并且用“黑”“白”名单进行过滤。  
白名单很容易理解，用集合是否为空和集合的contains()；  
黑名单是判断Float.isNaN(similarityValue)，因为此前在步骤11的输出时黑名单的similarityValue被设置为了Double.NaN。  
  
对于非booleanData，是用pref和相似度矩阵的PartialMultiply得到推荐度的值来进行排序。  
而booleanData的pref值都是1.0f，所以去计算矩阵相乘的过程没有意义，直接累加相似度的值即可。  
用这个数据排序就可得到推荐结果。  
输出为userid,RecommendedItemsWritable，后者实际是List<RecommendedItem<itemid,pref>>，  
这里的pref是相似度矩阵的PartialMultiply或是相似度累加计算出来的值而非实际值。  
  
**后注：**  
以上提到的FastIDSet，SequentialAccessSparseVector，RandomAccessSparseVector等等数据结构，  
是Mahout提供的一些大数据量存储和处理的一些高效实现，  
针对数据的特点而做的有针对性的优化，同时解决性能和空间的问题。  
在Mahout in Action的讨论CF和Cluster等的“数据的表达”章节中有专门的阐述，此处不再详细解释。

mahout的taste框架是协同过滤算法的实现。它支持DataModel，如文件、数据库、NoSQL存储等，也支持Hadoop的MapReduce。这里主要分析的基于MR的实现。

基于MR的CF实现主要流程就在org.apache.mahout.cf.taste.Hadoop.item.RecommenderJob类中（注意mahout有两个RecommendJob，要看清楚是哪一个包）。这个类的run方法就包含了所有的步骤。从上到下，完整的其实有10步（中间计算item相似度其实拆分成了3个job，我们也当做是一个phase吧）， 也就是说，如果指定了所有的必要参数，运行一次item-based CF算法，会执行12个JOB，当然有的步骤是可以忽略的，下面会讲。以下就是详细的每一步骤的分析：

**phase1: itemIDIndex**

这步主要是将itemId转成一个int。这里设计上其实有点小问题，如果item的数量非常多，比如超过int的最大值，那就有可能会出现重合了。所以用long其实更为合适。

   input:用户评分文件（这也是我们最原始的输入了），格式一般为：userId t itemId t score。注意输入必须是textfile的。可能是为了方便测试吧，mahout的很多包默认输出都是textfile格式的。

   map:(index, itemId)

   reduce: (index, itemId)

**phase2: toUserVector**

   input:用户评分文件

   param: --userBooleanData如果这个参数为true，则会忽略评分列，对于买或不买这类数据有时需要指这定这个值。

   map: (userId, itemId,pref)

   reduce: 以用户为key，输出成向量形式è (userId, VectorWritable<itemId, pref>)

**phase3: countUser**，计算用户数

   map: (userId)

   reduce: 输出总用户数count

**phase4: maybePruneAndTranspose**

   input: phase2的输出：userVector

   param: --maxCooccurrences

   map: (userId,Vector<itemId, pref>) è(itemId,DistributedRowMatrix<userId,pref>)，注意如果指定了—maxCooccurrences参数，这里会有裁剪， 每个userId最多对maxCooccurrences的itemId打分

   这里的DistributedRowMatrix，分布式行矩阵：行：itemId, 列：userId

   reduce: (itemId, VectorWritable<userId,pref>)

**phase5: RowSimilarityJob**

这一步比较关键，计算item相似度，它拆分成了三个JOB。

   param: --numberOfColumns, --similarityClassname,--maxSimilaritiesPerRow(默认:100)

**job1:weight**

      input:phase4的输出

       map: (itemId, VectorWritable <userId, pref>) ==>(userId, WeightedOccurrence<itemId, pref, weight>)

       这里的weight，对于欧氏向量距离，或者Pearson距离等，均为Double.NaN，即无效。在LoglikelihoodVectorSimilarity中有用到weight的值。

       reduce:(userId, WeightedOccurrenceArray<itemId, pref, weight>)

**job2:pairwise similarity \*item相似度计算\***

      map: 对同一用户的所有item-rating，输出两两item之间的关系 ==>(WeightedRowPair<itemA, itemB, weightA, weightB>, coocurrence<userId,valueA, valueB>) (同上，这里的权重weightA,B对于欧氏距离等可以忽略)

       reduce: 在这端，以<itemA,itemB>为key聚合了来自不同map的**所有用户**的打分，最后输出itemA和B的对称相似度（即以itemA为key或以itemB为key）==>(SimilarityMatrixEntryKey<itemA,similarity>, MatrixEntryWritable<WeightedRowPair<itemA, itemB,weightA, weightB>>) ，(SimilarityMatrixEntryKey<itemB,similarity>, MatrixEntryWritable<WeightedRowPair<itemB, itemA,weightB, weightA>>)

**job3:entries2vectors \*汇总item的相似items\***

param: --maxSimilaritiesPerRow

      map: (itemA, itemB, similarity) & (itemB,itemA, similarity) 这里在group的时候按相似度降序做了排序，如果有--maxSimilaritiesPerRow参数，则会做裁剪。

      reduce: (itemA, VectorWritable <item,similarity>)

至此，item相似度计算完毕。

**phase6: prePartialMultiply1**

   input: phase5的最后输出(即item相似度)

   map: 直接输出item对应的相似items，这里用VectorOrPrefWritable做了封装，表明有可能是相似度向量，也有可能是对item的打分，并且对item为自己的，将相似度设为Double.NaN，以过滤自身。è(itemId,VectorOrPrefWritable<item, similarity>)

    reduce: IdentityReducer

**phase7: prePartialMultiply2**

   input: phase2的输出userVectors

   map: 输出：(itemId, VectorOrPrefWritable<userId, pref>)

   这里默认考虑用户对10个item的评分，可以通过maxPrefsPerUserConsidered参数调整。

   如果指定了usersFile，则在setup时会把所有的userId读入内存，用于过滤。如果map输入数据的userID不在usersFile中，则会被忽略。注意，这是mahout的设计bug，对于比较大的数据集，很有可能造成OOM（事实上在我的测试程序中已经出现OOM了…），这种bug下面还会出现。输出的是用户的评分，同phase6的VectorOrPrefWritable的封装。

   reduce: IdentityReducer

**phase8: partialMultiply**

     input: 6和7的输出：prePartialMultiply1, prePartialMultiply2

     map: Identity。由于6和7的输出的key均为itemId，因而在reduce端同一item的相似item以及对应的用户评分会聚合到一起。

     reduce:(itemId, VectorAndPrefsWritable<similarityMatrix, List<userId>,List<pref>>) 没做特殊处理，直接合在一起，输出相似度矩阵，所有的userId及对item的打分。

**phase9: itemFiltering**

    将过滤文件输出成<userId, itemId>。如果指定了--filterFile参数，则在最后的聚合推荐中会过滤userId对应的items。这一步在实际中多数是可以忽略的，只要不指定这个参数即可。

**phase10: aggregateAndRecommend**

    map: 对每个用户，输出对当前item的评分，以及与当前item的所有相似itemsè(userId, PrefAndSimilarityColumnWritable<pref,vector<item, similarity>>)

   reduce: 聚合了这个用户所有的评分历史，以及相似items，计算对该用户的推荐结果 è (userId, List<itemId>)。

注意在reduce的setup中，会将phase1产生的所有itemId到index的映射读入内存，这里只要Item数据集稍大，就会OOM。这是比较严重的设计bug。

事实上，如果item是正规的整数，而不是guid之类的，phase1和这一步的读入内存是完全可以略掉的。这样的话就完全可以在企业级的数据集上使用（我的测试数据集是15亿+的user-item-rating，1.5亿+的用户，在最后这一步挂掉了，前面所有phase都能跑成功）。

至此，已经形成了推荐结果，CF完成。

以上的所有步骤中，phase5的计算item相似度是最慢的（这个其实也很直觉）