如何用5行SQL代码算出协同过滤的相似矩阵

对推荐系统有些了解的同学，都会知道协同过滤算法。而协同过滤算法的核心，就是相似矩阵。相似矩阵表示了任意两件商品之间的关联性。当用户购买一件商品时，就可以按相似度大小推送相关联的商品。对于多数推荐场景，都涉及到千万级的用户，在百万种的商品中的挑选，因此会有一个非常大的矩阵运算。如果直接计算，花费的时间是一个天文数字。而协同过滤本身是基于用户的历史行为，这样长的运算时间很难保证历史上的相似性在现在还有效。

1. 原理介绍

推荐最大的用处是帮助人们从千万种商品中发现自己喜好的商品。光凭借热度模型很难达到对每个人都有效。这样就需要更个性化对推荐算法。协同过滤基于的想法是寻找商品或者用户之间的关联。商品关联的依据主要是过去的用户行为，如果一个用户在两个商品上都有浏览或购买行为，则认为两个商品的相似度高一点。当有大量的用户行为数据沉淀，就可以计算两个商品间的相似度，而最常用的数学工具是余弦算法。它的几何意义，就是把每个商品用一个高维向量表示，每个用户算作一个维度，用户数就是商品的维度数，然后计算两个商品向量的余弦夹角。

这里可以看到，对于每两件商品，相似度的计算需要用到所有用户访问行为，因此计算次数是总的用户数。而所有商品间都需要计算，则要算商品数\*商品数次。如果是一千万用户在一百万商品上的推荐，计算量是107 \* 106 \* 106 = 1019次，用现在的GHz的CPU，也需要1010秒，即几年时间。因此必须要新的方法提高运算效率。

2. 目前的解决方法

对于相似矩阵浩大的运算量，最直接的想法就是做一些程序上的优化。我的前同事是这方面的高手，熟练切换C，C++，CPython等开发语言，提高了50%的速度。这样固然快了很多，但是还是远远不够满足业务的需求。之后，他对每个商品下的用户ID做了一个排序，计算内积时候按照排序好的用户ID索引。考虑到对于绝大多数商品，用户是没有交互的，计算时间降为了O(m1+m2)，其中m是一个商品上的实际用户数，这个数多在几百到几千，远小于千万。再将一些中间结果保存，避免重复计算，这些优化都做了以后可以将计算时间降低到几天时间，可以说在当时是一个很大的提升。

还有一些方法，也在尝试通过一些trade-off缓解计算时间问题，但是都没有真正解决。比如近似矩阵计算[1]，虽然时间提高，但是结果准确度下降。还比如增量矩阵计算[2]，在第一次运算时用时较长，之后进行增量计算，速度可以大大降低。

3. Hive-SQL实现方法

之后我借鉴别人经验，想到一种Hive-SQL的方法，在我前同事基础上，又实现了100倍以上的速度的提升，将总运算时长缩短到103秒，多次线上实验都是在十几分钟。有编程经验的人都知道，如果想提升50%的速度，只要在编程语言上花心思，但是如果想有100倍的提升，就需要在整个计算流程上进行重构。上一节提到，我前同事利用商品上用户行为的稀疏性提升的100倍的速度，将运算时间从 年级缩短到天级。在此基础上，我认识到商品之间的关联也是稀疏的，即绝大多数商品之间是不相关的，相比于百万级的候选集，与每个商品真正相关的不过百余件。因此，绝大多数商品之间是不需要计算相关度的。我们只需要，快速定位那些有可能相关的商品对，即有共同的用户行为的商品对。这个问题，估算一下其实很好理解。百万106商品间的交互次数是1012次，而商城里千万107用户沉淀下的行为最多是109次。这中间一个用户购买两次且在不同商品上的就更少。而这个数量是很难填满1012的格子。绝大多数的空格子，并不需要计算。

另外一个窍门是，这里最核心的Hive-SQL join语句在MapReduce机制下能够高速运算。MapReduce本身对所有的key作index，在join的时候可以自然分发，不需要重建索引，速度提高很多。

4. 优点和效果

当处理时间快50%，对分析师的益处是节省了工作时间，但如果快了10000倍，就是生产力的巨大革命。因为相似矩阵的生成速度降到了10分钟级，我们可以做到近实时计算相似矩阵。头30分钟的售卖结果可以马上用来运算相似矩阵，为下一时刻的推荐所用。而且，因为速度很快，协同过滤技术被延伸到了多个场景下，除了商品关联，还有品牌关联，品类关联，标签关联，搜索词关联等等，充分释放了用户行为数据的价值，发挥了协同过滤的威力。更且，因为在SQL环境中开发，所有掌握SQL而不会其他编程语言的分析师都可以使用，数据不落地，细节修改简易，为维护和调试算法提供了巨大的便利。

5. 延伸

讲到这里扩展一下，就是这个框架不仅仅适用于余弦相似度，对于其他相似度定义，同样可以通过若干行SQL代码解决。比如很常用的Jaccard相似度，在特定场合有效果的Log-Likelihood Ratio[3]，和一个最近在国内比较火的热扩散算法[4]，因为涉及到用商品根据用户交互计算关联度，都可以有很大程度的运算提高。

后续可能还有进一步提高的方法，比如用到Spark，或者在语句上压缩时间，欢迎读者探索。

文献:

[1] Dimension Independent Similarity Computation by Zadeh et al., <https://arxiv.org/abs/1206.2082>

[2] Incremental Collaborative Filtering for Highly-Scalable Recommendation Algorithms by Papagelis et al. , <https://www.ics.forth.gr/isl/publications/paperlink/LNCS_Formatted_ISMIS-05_34880553.pdf>

[3] *Accurate Methods for the Statistics of Surprise and Coincidence* by Ted Dunning, <http://aclweb.org/anthology/J93-1003>

[4] *Solving the apparent diversity-accuracy dilemma of recommender systems* by Zhou et al., <http://www.pnas.org/content/107/10/4511.full.pdf>

代码：

CREATE TABLE tbl\_user\_item\_cnt

AS SELECT userID AS userID,

itemID AS itemID,

IF(COUNT(\*) <= 25, COUNT(\*), 25) AS cnt

FROM tbl\_user\_item

GROUP BY userID,

itemID

HAVING userID <> ''

AND itemID <> ''; --- remove empty IDs ---

CREATE TABLE tbl\_item\_len

AS SELECT query,

COUNT(\*) AS cnt,

SQRT(SUM(cnt\*cnt)) AS len

FROM tbl\_user\_item\_cnt

GROUP BY query

HAVING cnt > 5; --- remove unpopular items, which have < 5 users ---

CREATE TABLE tbl\_user\_item\_vec

AS SELECT t1.userID,

t1.userID,

t1.cnt,

t2.len

FROM tbl\_user\_item\_cnt t1,

tbl\_item\_len t2

WHERE t1.itemID = t2.itemID;

CREATE TABLE tbl\_dot\_product

AS SELECT t1.userID AS userID1,

t1.itemID AS itemID1,

t1.cnt AS cnt1,

t1.len AS len1,

t2.userID AS userID2,

t2.itemID AS itemID2,

t2.cnt AS cnt2,

t2.len AS len2

FROM user\_item\_vec t1,

user\_item\_vec t2

WHERE t1.uid = t2.uid;

CREATE TABLE tbl\_similarity

AS SELECT itemID1,

itemID2,

SUM(cnt1\*cnt2)/(len1\*len2) AS similarity

FROM tbl\_dot\_product

WHERE itemID1 <> itemID2

GROUP BY itemID1,

itemID2,

len1,

len2

ORDER BY itemID1,

similarity DESC;

How to calculate the similarity matrix of collaborative filtering in 5 lines SQL code

For those who know about recommender system they definitely know the classic Collaborative Filtering (CF) algorithm. The core of the CF algorithm is the similarity matrix. The similarity matrix records the relevance between any two items. When a user buys a product, the relevant product is recommended according to the similarity matrix. However, in most scenarios where the recommender engine is used there are millions of users and hundreds of thousands of products, the computing of similarity matrix is very expansive. It can take years if it is computed by brute force. We cannot afford that time cost and a better way is expected.

1. Introduction

The recommender engine becomes popular in last a few years because it can find the product preferred by the customer from millions of choices. This is realized by a personalized recommendation algorithm. Collaborative filtering is based on the idea that products co-browse and/or co-buy by a customer are more likely to be related. By considering millions of co-browses and co-buys, we can extract the relationship among millions products. In the mathematic terms, every product is a vector of m dimension, where m is the number of customers. Usually, it is a vector of very high dimension and very sparse (meaning most values are 0). We normally measure the similarity by cosine, which is the angle between two product vectors.

Here you can see that for every two items, the cosine similarity calculates the dot product of two item vectors, which scans through all users. Also, to cover all product pairs, it calculates the number of goods \* goods times. If there are 10 million users and 1 million items, the calculation takes 107 \* 106 \* 106 = 1019 times. With the state-of-the-art GHz CPU, it needs 1010 seconds, or, a few years. We need a better method to improve the efficiency of the operation.

2. Some solutions

For such a vast amount of computations, the most straightforward idea is to do some procedural optimization. My former colleague is a master of this, who is skilled in C, C + +, CPython and many other development languages, and successfully increased the speed by 50%. This is certainly much faster, but still far from the requirement of the business. After that, he sorted user ID under each item, and the inner product is indexed according to the sorted user ID. Taking into account the vast majority of goods with no user interaction, the calculation time reduced to O(m1 + m2), where m is the actual number of users for each item, and usually in the number of a few hundred to a few thousands, far less than 10 million. In addition, by saving some intermediate results, the entire calculation can be reduced to a few days, which is a big improvement.

There are some other ways to improve the speed, with some trade-offs, for example, the approximate matrix calculation [1], where the time is improved by the cost of accuracy; also, incremental matrix calculation [2], where the first operation takes a while, followed by incremental calculation, and the speed can be greatly reduced.

3. Hive-SQL implementation

Inspired by some other’s work, I came up a Hive-SQL approach, which achieved another 100 times speed up comparing to my former colleague - in total the processing time is shortened to 103 seconds, and multiple online experiments were finished around 10 minutes. If you are a seasoned programmer, you know that to increase speed of 50%, you can fine tune the code, but if you want to have 100 times speed up, you need to rethink the entire calculation process. As mentioned in the previous section, my former colleague took advantage of the sparseness of user behaviors on products to speed up 100 times. On top of that, I realized that the connections between products are also sparse, that is, the vast majority of products are not related. Among millions of candidates, one is generally related to no more than a few hundreds. Thus, there is no need to calculate the correlation between most product pairs. What we need is to locate those pairs that are related, and in CF algorithm, is to calculate those with user co-behaviors. Let us do some simple algebra: millions of products potentially generate 1012 pairs, and for an e-Commerce site with 107 users in maximum has 109 actions, and less for a user of two actions on different items. To fill a 1012 grid with 109 numbers, clearly most cells are left open.

Another trick is that the Hive-SQL ‘join’ statement is very fast under the MapReduce mechanism. MapReduce hashes all keys for indexing, which saves the time for join operation to rebuild index, which speeds up a lot.

4. Implications

If an analytical process can be 50% faster, analysts are happy to leave work earlier, but if the process can be 10,000 times faster, analysts need to rethink their jobs. Now the similar matrix only takes 10 minutes to generate, which opens up the door for us to do near-line (between offline and online) computation. Think about this, we can use last 30 minutes’ sales results to calculate an up-to-date similar matrix for next second’s recommendation. Moreover, because of the speed and ease to use, collaborative filtering technology is extended to a number of use cases. Besides products association, it can calculate brand association, category association, tag association, search query association, etc., which fully releases the power of user behavior data. Furthermore, because it is in SQL environment, anyone knows SQL can leverage on this technology. Data preparation can be assembled along with the algorithm in the same pipeline, and maintenance and debugging algorithms are trivial tasks.

5. Next step

A number of extensions can be done on this framework. First, it is not only applicable to the cosine similarity, but other similarity definitions, such as the Jaccard similarity, the Log-Likelihood Ratio [3], and the heat emission algorithm [4], all by a few lines of SQL. Also, the rise of SparkSQL may also bring this framework to real-time, which takes advantage of concurrent learning to the CF algorithm.

Literature:

[1] Dimension Independent Similarity Computation by Zadeh et al., <https://arxiv.org/abs/1206.2082>

[2] Incremental Collaborative Filtering for Highly-Scalable Recommendation Algorithms by Papagelis et al. , <https://www.ics.forth.gr/isl/publications/paperlink/LNCS_Formatted_ISMIS-05_34880553.pdf>

[3] *Accurate Methods for the Statistics of Surprise and Coincidence* by Ted Dunning, <http://aclweb.org/anthology/J93-1003>

[4] *Solving the apparent diversity-accuracy dilemma of recommender systems* by Zhou et al., <http://www.pnas.org/content/107/10/4511.full.pdf>

代码：

CREATE TABLE tbl\_user\_item\_cnt

AS SELECT userID AS userID,

itemID AS itemID,

IF(COUNT(\*) <= 25, COUNT(\*), 25) AS cnt

FROM tbl\_user\_item

GROUP BY userID,

itemID

HAVING userID <> ''

AND itemID <> ''; --- remove empty IDs ---

CREATE TABLE tbl\_item\_len

AS SELECT query,

COUNT(\*) AS cnt,

SQRT(SUM(cnt\*cnt)) AS len

FROM tbl\_user\_item\_cnt

GROUP BY query

HAVING cnt > 5; --- remove unpopular items, which have < 5 users ---

CREATE TABLE tbl\_user\_item\_vec

AS SELECT t1.userID,

t1.userID,

t1.cnt,

t2.len

FROM tbl\_user\_item\_cnt t1,

tbl\_item\_len t2

WHERE t1.itemID = t2.itemID;

CREATE TABLE tbl\_dot\_product

AS SELECT t1.userID AS userID1,

t1.itemID AS itemID1,

t1.cnt AS cnt1,

t1.len AS len1,

t2.userID AS userID2,

t2.itemID AS itemID2,

t2.cnt AS cnt2,

t2.len AS len2

FROM user\_item\_vec t1,

user\_item\_vec t2

WHERE t1.uid = t2.uid;

CREATE TABLE tbl\_similarity

AS SELECT itemID1,

itemID2,

SUM(cnt1\*cnt2)/(len1\*len2) AS similarity

FROM tbl\_dot\_product

WHERE itemID1 <> itemID2

GROUP BY itemID1,

itemID2,

len1,

len2

ORDER BY itemID1,

similarity DESC;