Online Collaborative Filtering

Joshua Morton

May 2014

Contents

1 Introduction

This outlines some of the mathematics used in the online collaborative filter used in other areas of this tool. This is both for my sanity and a way to outline and think through some of the mathematics used elsewhere in the codebase. To begin with, I'll outline the collaborative filtering code as a whole.

The estimated opinion, r, by user u of an item i is defined below, where U is the set of all users that are not u.

$$r_{u,i} = k \sum_{u' \in U} simil(u, u^{'}) r_{u,i}$$

This function makes use of two additional functions, $simil(u, u^{'})$ and k. $simil(u, u^{'})$ operates over the set I, which is defined as the intersection of the sets of objects that have previously been rated by both u and $u^{'}$

$$k = \frac{1}{\sum\limits_{u' \in U} |simil(u, u')|}$$

$$simil(u, u^{'}) = \frac{\sum\limits_{i' \in I} (r_{u,i})(r_{u',i})}{rss(u) \times rss(u')}$$

In this case, rss signifies the root sum squares of all ratings for that user. Here, S refers to the set of all items rated by a given user. To put this mathematically:

$$rss(u) = \sqrt{\sum_{i \in S} r_{u,i}^2}$$

Putting this all together, we get that the formula to calculate the rating for a given item at any time can be expressed as

$$r_{u,i} = \frac{\sum\limits_{u' \in U} (simil(u, u^{'}) \cdot r_{u,i})}{\sum\limits_{u' \in U} |simil(u, u')|}$$

This expands to the awful looking and disgusting piece of math that is

$$r_{u,i} = \frac{\sum\limits_{u' \in U} \left(\frac{\sum\limits_{i' \in I} (r_{u,i})(r_{u',i})}{\sqrt{\sum\limits_{i \in S} r_{u,i}^2} \sqrt{\sum\limits_{i \in S} r_{u',i}^2}} \cdot r_{u,i} \right)}{\sum\limits_{u' \in U} \left| \frac{\sum\limits_{i' \in I} (r_{u,i})(r_{u',i})}{\sqrt{\sum\limits_{i \in S} r_{u,i}^2} \sqrt{\sum\limits_{i \in S} r_{u',i}^2}} \right|}$$

2 An Example

Much of this is repetitive code and can be made into functions, but this is still obviously worst case $\mathcal{O}(n^2)$ to calculate. That cannot be changed, but through some clever mathematics, the speed with which an opinion can be calculated can be heavily improved, using a bit of magic.

To do this, the first step is to provide a small scale example of what happens within the code, in this case on a 4×4 matrix/table that will be used in the example. This can then be generalized to matrices (and tables) of arbitrary size. To begin with, here is an example table of users and ratings:

	A	B	C	D
W	A,W	B,W	C,W	D,W
X	A,X	B,X	C,X	D,X
Y	A,Y	B,Y	C,Y	D,Y
Z	A,Z	B,Z	C,Z	D,Z

In this table, Users are represented by W, X, Y, and Z. Items are represented by A, B, C, and D. Therefore User W's rating for item A would be found in position AW. No matter the database structure used, the resulting data can be essentially expressed in this manner. Keep in mind that this matrix may be sparse and as a result, the value at any given position may be null.

On this note, during proceeding examples that use values, a cell value of 0 will represent a null, meaning that the given user has no existing opinion on the item in question. Valid values will be from 1-5, so any given space can have an integer value 0, 1, 2, 3, 4, or 5. The results of the Collaborative Filter may not be integers, and will be expressed as floats so as not to lose accuracy.

The beginning matrix we will use is this:

$$\begin{bmatrix} 1 & 2 & 3 & 5 \\ 1 & 0 & 3 & 5 \\ 1 & 1 & 1 & 1 \\ 5 & 5 & 5 & 5 \end{bmatrix}$$

In this matrix, if like before users are represented by W, X, Y, Z and items by A, B, C, D we are looking for (B, X), which is user X's opinion of

item B. To calculate this we first highlight the item for which we are trying to predict a rating

$$\begin{bmatrix} 1 & 2 & 3 & 5 \\ 1 & \boxed{0} & 3 & 5 \\ 1 & 1 & 1 & 1 \\ 5 & 5 & 5 & 5 \end{bmatrix}$$

The next step is to calculate the similarities between users. To do this, the algorithm takes the other items of user X and compares the ratings that X gave them to the ratings that W, Y, and Z gave. So for each item A, C, and D the algorithm compares the values for each user via the second part of the $simil(u, u')$ function. So the result is this:

$$simil(X, W) = \frac{1+9+25}{\sqrt{(1+9+25)}\sqrt{(1+9+25)}} = \frac{35}{\sqrt{35}\sqrt{35}} = 1 \quad (1)$$

$$simil(X, Y) = \frac{1+3+5}{\sqrt{(1+1+1)}\sqrt{(1+9+25)}} = \frac{9}{\sqrt{3}\sqrt{35}} = .878... \quad (2)$$

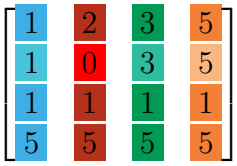
$$simil(X, Z) = \frac{5+15+25}{\sqrt{(25+25+25)}\sqrt{(1+9+25)}} = \frac{55}{\sqrt{75}\sqrt{35}} = .878... \quad (3)$$

simil(X, W)	1
simil(X, Y)	.878
simil(X, Z)	.878

This example was chosen specifically to illustrate a few facts about the algorithm. First, The $simil(u, u')$ function essentially finds (a version of) linear distance between to values in some n-dimensional space. As a result, whether or not the related values are higher or lower, the distance will be unchanged (as long as they are equally different). This is good because it means that there is not any kind of bias introduced based on the higher and lower values.

Now that we have the similarity values, there are two finals steps, to calculate k and to calculate $r_(u, i)$. Calculating k can be considered part of the same step as calculating the similarities, because it is simply $1/\sum(simil(u,u'))$. Both of these use the same values.

$$k = \frac{1}{\sum\limits_{u' \in U} |simil(u, u')|} = \frac{1}{1 + .878 + .878} = .363$$



The final step is then to propagate back down, using the rating and k value to calculate the final opinion. In this case, the darker values have been compared to the lighter values to get a weighting, and this weighting has been used to calculate weights for the dark red columns which are then used to calculate the expected value for the final, unknown position.

$$r_{u,i} = k \sum_{u' \in U} simil(u, u')r_{u,i} = .363 \times (1 * 2 + .878 \times 1 + .878 \times 5) = 2.638$$

It is worth mentioning that this algorithm does function on less dense matrices and datasets.

3 An Analysis of Operations

In its current form, any time you wish to take data from the table, you need to calculate this $\mathcal{O}(n^2)$ (technically, its $m \times n$, but in this case we assume that m and n are of similar size. This may not be the case, but it works for the example.) operation. This is not a great situation, as for medium to large datasets (say, $n = 10000$), the algorithm can become slow since, even with some steps taken to increase the algorithm's efficiency, 100,000,000 operations for each database query is a tad steep.

There are multiple ways to combat this. A common approach is to use a k-nearest neighbor based approach. This uses a linear algorithm to calculate the k-nearest neighbors to any given user and only runs in $\mathcal{O}(n \times k)$ where k is a value of your choice, a significant improvement. The trade off is that there is a possible loss of accuracy depending on the size of the dataset. This loss is probably small, but in sparser, smaller datasets will be large enough to possible cause problems.

In this I outline an alternative method. By using a combination of threaded queues and multilevel caches, you can keep completely accurate information and calculate, update, and change values in significantly less than $\mathcal{O}(n^2)$ time. To begin, there are a number of functions that a useful Collaborative Filter must implement. You must be able to **Predict an Opinion**, **Change an Opinion Get a Known Opinion**, **Add a New Opinion** and **Remove an Old Opinion**.

The downside to the method outlined here is that in some cases you will have slightly outdated information, depending on the number of changes happening to the database during your query. This would be the case no matter what, since at scale caching is necessary anyway, but this problem may be exacerbated by the method outlined.

3.1 The Data Structure

The collaborative filter is built as a series of stacked matrices. These can be implemented in any way, either as caches in memory, persistent databases, etc. The examples work off of a model where each matrix is a cache in memory implemented as a two-dimensional hash-table. These caches start empty, and are filled partially as calls are made to the database. In other words, when the structure is created, the tables are empty, but after a call for user **U**, **cache**[**U**] would be filled.

To reiterate, this is by no means the only way to implement such a cache. They could be created and fully filled at creation, but this was, in my opinion, a good method that had only minor losses in either space or time efficiency.

There are three levels of matrices. The first one is the **Opinion Matrix**. This Matrix contains the concrete opinions that people hold. It would be implemented with the structure Map(User → Map(Item → Rating)).

The second matrix, referred to as the **Similarity Matrix** contains calculated similarities between users. It has the structure Map(User → Map(User → Similarity)).

The final matrix contains the predicted opinions that a user would have for an item and is called the **Calculated Matrix**. It is structured as Map(User → Map(Item *to* Rating)), similar to the first matrix, but the calculated values will differ from the actual values and the matrix has an arbitrary sparsity, as opposed to being only as full as the users provide information.

3.2 Get a Known Opinion

This is entirely unchanged, much like normal, this simply requests the opinion from the lowest matrix or returns a null value if the item is not in the matrix. Nothing difficult here.

3.3 Predict an Opinion

This too works similar to the example. The main difference is that calculations become a fallback and instead the algorithm attempts to get relevant information from higher-level matrices if possible. For example, instead of calculating user A's similarity to user B, the algorithm will first check the **Similarity Matrix** before actually dropping down and calculating the values itself. In python pseudocode, this might look like this:

```
def calculateOpinion(user, item):
  if item in user in CalculatedMatrix:
    return CalculatedMatrix[user][item]
  else:
    //calculate the result from this
    sum(similarity(user, other) for other in users)

def similarity(user, other):
```

```
  if other in user in SimilarityMatrix:
    return SimilarityMatrix[user][other]
  else:
    //calculate the similarity value as outlined
    sum(rating(user, item) * ... for item in items)

def rating(user, item):
  reutrn OpinionMatrix[user][item]
```

The important thing to note (other than my unnatural love for generator expressions) is that in each minor function, the calculation of the opinion, similarity, and rating, you can first search a cache. This means that when calculating an opinion, if the opinion is known, the operation becomes $\mathcal{O}(1)$. If all similarities between users are known, the operation becomes $\mathcal{O}(n)$. This is a well known method of dynamic programming. The difficult portion is in keeping the caches updated and keeping clashes from occurring.

3.4 Change an Opinion

The method for changing an opinion is not easy. With a cache free variant, it is simple, you change the opinion and moving forward, all newly calculated ratings use the new opinion. Indeed, this setup could be easily changed to work similarly, simply by updating the lowest level matrix when an opinion changes and then removing any similarity values and calculated ratings that propagated from that user. The problem is that this would invalidate most, if not all, calculated ratings any time the Opinion Matrix changed, making the system significantly less useful.

Instead, lets talk for a second about the similarity between two users. Given two users x and y, who have opinions on 4 items, there is a similarity value between them. This similarity value can be expressed as

$$simil(x,y) = \frac{r_{x,a}r_{y,a} + r_{x,b}r_{y,b} + r_{x,c}r_{y,c} + r_{x,d}r_{y,d}}{\sqrt{r_{x,a}^2 + r_{x,b}^2 + r_{x,c}^2 + r_{x,d}^2}\sqrt{r_{y,a}^2 + r_{y,b}^2 + r_{y,c}^2 + r_{y,d}^2}}$$

This can be generalized as necessary. What it means is that for any given pair of people, their similarities can be unpacked into a set of values. When one singular value changes, you can use the unpacked form and change only a small number of values instead of recalculating everything from nulls.

The problem with this is that you cannot simply store $simil(u, u^{'})$. Instead, you need to store it in a special form as a non-reduced fraction. This is because you lose some information when you divide the top portion by the bottom portion, you cannot unpack without making database calls and multiple calculations. This is actually quite simple, it requires storing similarity not as a single number, but as two separate numbers.

$$simil(u, u^{'}) = \frac{\sum_{i' \in I} (r_{u,i})(r_{u',i})}{rss(u) \times rss(u')} = \frac{a}{b}$$

In this situation, a and b represent the top and bottom of the fractional value of the similarity. The user x has changed their opinion of item i. The old opinion will be called o and the new one n. To change a from the old value to the new one, you simply take $a_{new} = a_{old} - o \times r_{y,i} + n \times r_{y,i}$ Refactoring you get $a_{new} = a_{old} + r_{y,i}(n - o)$. This takes only $\mathcal{O}(1)$ time, two database calls, one for o and one for $r_{y,i}$.

Calculating Δb is a tad more difficult. Much like before, where you had $\frac{a}{b}$ and lost information, in the multiplication of $rss(x) \times rss(y)$ there is once more a loss of information. By storing both of those values, $rss(x)$ and $rss(y)$ we avoid any problem. $rss(y)$ remains unchanged and can be gathered via database call, and to calculate $\delta rss(x)$ the following works:

$$rss(x)_{new} = \sqrt{rss(x)^2_{old} - o^2 + n^2}$$

This too requires only a constant number of database calls. It is important to note that the rss is constant and needs only be stored once for any given person, while a is dependent on each pair of users and so needs to be