Implementing a Rating-Based Item-to-Item Recommender System in PHP/SQL

Daniel Lemire
Université du Québec

Sean McGrath
University of New Brunswick

November 4, 2005

Abstract

User personalization and profiling is key to many succesful Web sites. Consider that there is considerable free content on the Web, but comparatively few tools to help us organize or mine such content for specific purposes. One solution is to ask users to rate resources so that they can help each other find better content: we call this rating-based collaborative filtering. This paper presents a database-driven approach to item-to-item collaborative filtering which is both easy to implement and can support a full range of applications.

1 Introduction

There is no question that the Web is now so rich that we are limited not by its content, but by the quality of the tools we use to search content. For example, Project Gutenberg [3] has several Gigabytes of free books one can download while inDiscover [6] has hundreds of free songs (MP3s) one can download for free. Many e-Commerce sites have equally extensive content. It is sometimes very difficult to organize items in an easily browsable fashion and thus, we expect Web sites to offer meaningful recommendations. However, such recommendations should not be limited to best selling titles or content already known by the user.

On the other hand, users will rate items when given a chance as web sites such as RatingZone, Amazon, and Epinions demonstrate. Almost all Web content can be rated: from images to blogs. Note also that ratings can be as sophisticated as needed: we can rate items on different attributes and take into account the context of the rating. As many users enter ratings, it is possible to predict how a given user would have rated something unknown to him by a comparison with other users.

Technical Report D-01, January 2005 (the report was edited on November 4th 2005 following comments by Chris Harris.

Implementing personalization in a Web site might not always be a challenge [2]. However, collaborative filtering, that is, personalization based on a database of user preferences, is more difficult because we need to infer unknow preferences. Amazon has a well known recommender system [5] which takes into account which book you are looking at to recommend a similar book. Since its model is based on relationships between any two books, and not on clusters of books or on relationships between users, we call it an item-to-item algorithm.

A few rating-based item-to-item algorithms have been proposed [9, 8]. Among those, this paper focuses on ther Slope One family [4] as they are especially convenient to implement. Like Amazon's algorithm, rating-based item-to-item algorithms are based on similarity accross items instead of a similarity accross users [7] which means that we can sometimes do more precomputation and accomodate a larger number of users. Also, item-to-item algorithms are particularly suited for "item similarity" applications.

The main contribution of this paper is to provide a practical implementation guide using commonly available and inexpensive tools (PHP and SQL) of an item-to-item recommender system. The guide is based on our experience designing a publicly available Bell/MSN Music Recommender System (http://www.inDiscover.net). We also briefly address scalability issues based on our practical experience.

2 A PHP/SQL Approach

In 2004, PHP was one of the 5 more popular programming languages overall and it one of the most popular Web programming language. Most PHP-based application use a relational database with a SQL interface. Hence, designing a PHP/SQL recommender system is an important problem. While we use the MySQL syntax, our minimalist approach to SQL will insure portability. The reader should note that it is possible to remove much of the PHP code we are using and have a pure SQL approach instead, but such an approach would be database vendor specific.

We assume numerical ratings, say from 0 to 10, though the range is irrelevant, are collected for users (having an "userID") accross a range of items (having an "itemID"). Hence, you could have a SQL table defined follows:

```
CREATE TABLE rating (
    userID INT PRIMARY KEY,
    itemID INT NOT NULL,
    ratingValue INT NOT NULL,
    datetimestamp TIMESTAMP NOT NULL
);
```

In the following sections, we show how to predict user ratings based on this table. Given predicted ratings, it is easy to provide recommendations to users: simply search for the highest predicted ratings.

3 Precomputing Popularity Differentials

As stated in [4], we believe that the following characteristics are desirable in a recommender system:

- easy to implement and maintain: all aggregated data should be easily interpreted by the average engineer;
- updateable on the fly: the schemes should not rely exclusively on batch processing;
- not demanding from new users: as long as we can guess how a use feels about one item, we should immediately be able to offer recommendations;
- efficient at query time: speed should not depend on the number of users and we should use extra storage as much as possible to precompute the queries;
- reasonably accurate: the schemes should be competitive with more expensive or complex schemes even though simplicity and convenience are our main concerns.

The WEIGHTED SLOPE ONE algorithm [4] meets these requirements. Other algorithms such as memory-based ones [1] do not allow us to precompute the recommendations and require that the user has rated several items before a sensible recommendation can be made.

SLOPE ONE is based on a simple "popularity differential" which we compute by subtracting the average rating of the two items. Considering only users who rated both A and B, say that there are 10 such users, we sum the ratings that A and B got, say 55 and 75. Clearly, B is rated higher than A by 2 on average.

We call the item-to-item matrix which results from this precomputation, that is, both the COUNT and SUM of the ratings for any pair of items, the *dev* table. Assuming that rating values are integers, say from 1 to 10, we can create a *dev* table as follows.

```
CREATE TABLE dev (
  itemID1 int(11) NOT NULL default '0',
  itemID2 int(11) NOT NULL default '0',
  count int(11) NOT NULL default '0',
  sum int(11) NOT NULL default '0',
  PRIMARY KEY (itemID1,itemID2)
);
```

Each time a new user rating is entered, we update the *dev* table. The fact that we can update the *dev* table online is a consequence of the fact that we store both the SUM and COUNT aggregates and not simply the average. Suppose that "$itemID" is the ID of the item just rated and "$userID" is the ID of the user who did the rating, the following PHP code updates the database accordingly.

```
// This code assumes $itemID is set to that of
// the item that was just rated.
// Get all of the user's rating pairs
$sql = "SELECT DISTINCT r.itemID, r2.ratingValue - r.
   ratingValue
           as rating_difference
           FROM rating r, rating r2
           WHERE r.userID=$userID AND
                   r2.itemID=$itemID AND
                   r2.userID=$userID;";
$db_result = mysql_query($sql, $connection);
$num_rows = mysql_num_rows($db_result);
//For every one of the user's rating pairs,
//update the dev table
while ($row = mysql_fetch_assoc($db_result)) {
    $other_itemID = $row["itemID"];
    $rating_difference = $row["rating_difference"];
    //if the pair ($itemID, $other_itemID) is already in
        the dev table
    //then we want to update 2 rows.
    if (mysql_num_rows(mysql_query("SELECT itemID1
    FROM dev WHERE itemID1=$itemID AND itemID2=
        $other_itemID",
    $connection)) > 0)  {
        $sql = "UPDATE dev SET count=count+1,
        sum=sum+$rating_difference WHERE itemID1=$itemID
        AND itemID2=$other_itemID";
        mysql_query($sql, $connection);
        //We only want to update if the items are
            different
        if ($itemID != $other_itemID) {
            $sql = "UPDATE dev SET count=count+1,
            sum=sum-$rating_difference
            WHERE (itemID1=$other_itemID AND itemID2=
                $itemID)";
            mysql_query($sql, $connection);
        }
    }
    else { //we want to insert 2 rows into the dev table
        $sql = "INSERT INTO dev VALUES ($itemID,
            $other_itemID,
        1, $rating_difference)";
        mysql_query($sql, $connection);
        //We only want to insert if the items are
            different
        if ($itemID != $other_itemID) {
                $sql = "INSERT INTO dev VALUES ($other_itemID
                    ,
                $itemID, 1, -$rating_difference)";
                mysql_query($sql, $connection);
            }
        }
}
```

We can use this *dev* table for two purposes: non-personalized and personalized recommendations. In both cases, the precomputed *dev* table allows us to provide recommendations online.

4 Non-Personalized Recommendations

A recommendation is "non-personalized" if it doesn't depend on a user profile. For example, a basic search engine providing a list of web pages makes a non-personalized recommendations. Another example is when all users browsing a given product on an e-Commerce web site see the same recommendations.

It is usually easy for a user to find one item of some interest on a given Web site through a list of popular items or through the search engine. From this point, it is useful to offer to the user other "related" items. Specifically, we want to help the user in finding items he likes as much or more than the current item. Notice that this approach is different from a similarity measure where we seek to present similar items: item-to-item algorithms are not necessarily based on similarity accross items.

In effect, given that the user is browsing item number 1, say, we are looking for items with ID "itemID2" such then:

- a sufficient large number of people (defined by a threshold) rated both item 1 and "itemID2";
- on average, we want "itemID2" to be rated as high as possible among users who also rated item 1.

If the *dev* table has been computed, then the following SQL code ranks all candidate items where we set the COUNT threshold to 2 and the current item (item being browsed) is $current_item. It is then a simple matter of showing part of this list to the user.

```
SELECT itemID2, (sum / count) AS average
FROM dev
WHERE count > 2 AND itemID1 = $current_item
ORDER BY (sum / count) DESC
LIMIT 10
```

5 Personalized Recommendations

When it comes time to generating a list of recommendations we need to define a prediction function to predict how the current user would rate an item (see below). Given this function, it is a simple matter to run this function over a large set of items in order to determine which items are most likely to be interesting to the user.

```
function predict($userID, $itemID) {
    global $connection;
    $denom = 0.0; //denominator
    $numer = 0.0; //numerator
    $k = $itemID;
    $sql = "SELECT r.itemID, r.ratingValue
    FROM rating r WHERE r.userID=$userID AND r.itemID <>
        $itemID";
    $db_result = mysql_query($sql, $connection);
    //for all items the user has rated
    while ($row = mysql_fetch_assoc($db_result)) {
        $j = $row["itemID"];
        $ratingValue = $row["ratingValue"];
        //get the number of times k and j have both been
            rated by the same user
        $sql2 = "SELECT d.count, d.sum FROM dev d WHERE
            itemID1=$k AND itemID2=$j";
        $count_result = mysql_query($sql2, $connection);
        //skip the calculation if it isn't found
        if(mysql_num_rows($count_result) > 0) {
            $count = mysql_result($count_result, 0, "
                count");
            $sum = mysql_result($count_result, 0, "sum");
            //calculate the average
            $average = $sum / $count;
            //increment denominator by count
            $denom += $count;
            //increment the numerator
            $numer += $count * ($average + $ratingValue);
        }
    }
    if ($denom == 0)
        return 0;
    else
        return ($numer / $denom);
}
```

One problem in the code presented above is that the “predict” function used for personalized recommendations has a great number of SQL calls which can be especially damaging if “predict” function is called often, over a wide range of itemIDs. An alternative is to use a single SQL call as follows: [1]

```
function predict_all($userID ) {
     $sql2 = "SELECT d.itemID1 as 'item', sum(d.count) as
          'denom',
     sum(d.sum + d.count*r.ratingValue) as 'numer'
     FROM item i, rating r, dev d
     WHERE r.userID=$userID
     AND d.itemID1<>r.itemID
     AND d.itemID2=r.itemID
     GROUP BY d.itemID1";
     return mysql_query($sql2, $connection);
}
```

Alternatively, if you are only interested in the best items[2], you could use a SQL query where you rank and select the best items:

```
function predict_best($userID, $n ) {
     $sql2 = "SELECT d.itemID1 as 'item',
     sum(d.sum + d.count*r.ratingValue)/sum(d.count) as '
         avgrat'
     FROM item i, rating r, dev d
     WHERE r.userID=$userID
     AND d.itemID1<>r.itemID
     AND d.itemID2=r.itemID
     GROUP BY d.itemID1 ORDER BY avgrat DESC LIMIT $n";
     return mysql_query($sql2, $connection);
}
```

One the other hand, if the number of items is large and storage space is an issue, one can limit the *dev* table only to item-item entries where the COUNT value is above a given threshold (say 1 or 2). Doing so can improve prediction performance and reduce storage tremendously, but one must then use a more complex update function since some entries are now missing (not precomputed).

In the worse case, complexity of all operations is linear in the number of items user can rate. If you have a really large number of items, it might be best to partition them, hence for a book recommender system, you could consider cookbook separately from novels. Also, if we ask users to rate several attributes, these different attributes can be in different tables.

[1] The text used to say “If one has ample storage and few items, but has need for very fast personalized recommendations, it is possible to populate the *dev* table densely: initialize the table with default value 0 for all entries. While arguably, a relational database is no longer the best tool, it is then easier to write an optimized predict function which computes all predictions in one step:” but Chris Harris pointed out that you don’t need to initialize the dev table to zero.

[2] This insight was contributed by Chris Harris.

6 Conclusion

We have presented the implementation of an item-to-item recommender system. In this system, one first precomputes popularity differentials in a matrix from which non-personalized and personalized recommendations can be computed online. Based on our experience with the inDiscover.net Web site, we can say that this approach gives sensible recommendations and is reasonably easy to implement..

This work was suppoted by NSERC (grant number 261437). We would like to thank Harold Boley and Bruce Spencer for valuable discussions.

References

[1] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Fourteenth Conference on Uncertainty in AI*. Morgan Kaufmann, July 1998.

[2] Dave Cartwright. Customise your content with user profiling. *Web Developer's journal*, 2000. http://www.webdevelopersjournal.com/articles/user_profiling.html.

[3] Michael Hart, Gregory Newby, and Marcello Perathoner. Project gutenberg. http://www.gutenberg.net/.

[4] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering. In *Proceedings of SIAM Data Mining (SDM'05)*, 2005.

[5] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, January 2003.

[6] Sean McGrath, Daniel Lemire, Harold Boley, Marcel Ball, and Bruce Spencer. indiscover.net. http://www.indiscover.net/.

[7] Paul Perry. A minimalist implementation of collaborative filtering: Automated collaborative filtering (acf) in sql. http://www.paulperry.net/notes/acf_sql.asp, November 2000.

[8] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl. Item-based collaborative filtering recommender algorithms. In *WWW10*, 2001.

[9] S. Vucetic and Z. Obradovic. A regression-based approach for scaling-up personalized recommender systems in e-commerce. In *WEBKDD '00*, 2000.