COMP3208 - Social Computing Techniques
Module Assignment

Azzino Tommy & Drago Matteo

Theoretical Introduction

A recommender system is a technology that involves two side of the same coin: *users* and *items*. Items could be products, movies, events, articles that are going to be recommended to users such as: customers, app users, travellers. We can relate the concept of recommendation to a real life situation: if you think to a small shop or boutique, then it's usual that a merchant knew personal preferences of everyday customers; his high quality advices satisfy clients increase profits and visibility of the shop (thank also to the usual word-of-mouth between people).
When dealing with online market places (Netflix, Amazon, Ebay or Asos), personal recommendations, suited up for each particular and different user, can be generated by something that can be seen as an "artificial merchant": the **recommender system**.
There are several definitions that we can find in the literature and in the *Internet*. Quoting Wikipedia:

> *"A recommender or recommendation system (sometimes replacing "system" with a synonym such as platform or engine) is a subclass of information filtering system that seeks to predict the rating or preference that a user would give to an item".*

In our opinion such definition is not completely correct; recommender systems are more than what stated. We can think about it as an *personal assistant*, because it can help you in order to make the right decision when you're going to buy something. Moreover, it knows your preferences and tastes; therefore it's not only a machine or an IT system: it perfectly suits you. A more proper definition could be:

> *"A recommender systems is a system that help users discover items they may like, based on a sort of personal knowledge of each user".*

Formally, such systems filter different items after studying a personal profile. They differ from many other information filtering systems, such as standard search engines and *reputation systems*, because recommendations are made on the specific user or group of users. We want to stress that the main goal for a recommender system is to show the user what he may like, guiding him on a platform that potentially offers a wide and heterogeneous set of choices. To achieve this purpose, the system can adopt different *user* or *item-based* models such as: the given rates to different items, preferences on distinct items typologies, either *demographic* or *context* information.
After the design of a good model, we need to find a way to elaborate this huge amount of information in order to extract the peculiarities between users and items that allows to achieve good recommendations.
Depending on the objective you want to fulfil, there are many different recommendation paradigms. The choice of the best paradigm depends on the type of data available and the computational efforts that one would like to employee when building the recommender system. For the latter point, computational time and power of available resources are the most important factors to be considered. In addition, talking about the different types of available data, a good idea would be to use hybrids systems that combines multiple different approaches in order to address and overcome each method's limitations.
There are four main paradigms of recommendation systems, which are described below:
Content-based: user's profile is automatically generated and updated using, for example, items already bought by the user. Such algorithms recommend similar items based on their description, focusing on items that are similar to each others according to certain particular features. The main advantage of content-based recommendation is that it does not require too much past history regarding one user or among users and items. In many everyday scenarios, this is the most common implemented approach. For example, personalized search is achieved by recommending web pages which are similar in content to those previously viewed by an user.
Knowledge-based: constructs a personal user profile using manually generated information about both users and items; this is done for example asking explicitly to a user questions about his preferences. Then, item similarity is decided by experts. This is the main limitation of this recommendation paradigms: we

need the presence of experts in the sector capable of determine the best recommendation for a user, starting from the profile obtained. Nonetheless, when recommending to first-time users, knowledge based becomes one of the strategy that gives best performances.
Social and demographic: these recommender systems suggest items that are liked by friends or friends-of-friends, or items liked by demographically-similar people. Furthermore, they don't need any preferences by the user, making them very powerful and easy to apply.
Collaborative filtering: this is the most popular technique involved in the construction of recommender systems, and it is used by many companies and e-commerce sites. The idea is to build a matrix that represents similarity between users or items. This matrix is then used for example to predict missing preferences for a certain item, of to recommend to the user items with the highest predicted rate. The straightforward advantage of this approach is that there's no need of content information for an item or the presence of experts that suggest a specific item to a particular user. Only a set of users and items together with a notion of *preference* are needed to implement collaborative filtering. Moreover, this technique has been studied and researched by an huge amount of experts, making it pretty well-understood.
In our work, we focused on collaborative filtering, which details that are given in the following sections.

MATLAB Approach

We decided to implement *collaborative filtering* with the *user-based* approach: under the assumption that people who exhibits same preferences in the past will likely do the same in future, we want to find a way to group these users, or to find a measure of similarity between them. One way to do it (which is also the one we adopted) is make use of the **jointly rated items**.
As a consequence, in order to get the similarity between two users we need first to collect all the items that both have reviewed. Then, using u_1 and u_2 to refer to the different users, and $\mathcal{I}$ for the set of jointly rated items, we can find the similarity using:

$$sim(u_1, u_2) = \frac{\sum_{i \in \mathcal{I}} (r_{u_1,i} - \bar{r}_{u_1})(r_{u_2,i} - \bar{r}_{u_2})}{\sqrt{(r_{u_1,i} - \bar{r}_{u_1})^2}\sqrt{(r_{u_2,i} - \bar{r}_{u_2})^2}} \tag{1}$$

where $\bar{r}_{u_j}$ is the average rate of the j-th user and $r_{u_j,i}$ is the rate given by the j-th user to the i-th item.
The measure just evaluated takes the name of Pearson Coefficient, which is demonstrated to obtain significant performances when it comes to collaborative filtering implementations; it takes value from -1 which correspond in complete uncorrelation (what we simply call in jargon "difference of tastes"), to +1 which on the contrary means that users have tastes exactly equal.
It should be clear that in this way we'll eventually obtain a matrix with only ones in the diagonal (every user is completely similar to himself). However, we cannot fill all the cells of the matrix: of course, when the set of jointly rated items is empty, similarity won't be computed. We'll delve more into the details of this later in this report, when we'll talk about the **cold start problem**.
Our first idea was to evaluate this matrix with MATLAB, given that it is naturally optimized for operations with matrices and vectors. Previous works with this programming language suggest us that, despite the dimension of the database, we could obtain significant and fast results.
For this problem, the training set contained 16045651 entries and was made of three columns: **userID**, **itemID**, **rating**; we imported everything into a `table` data structure (note that the CSV was ordered by userID). Then, in order to study if the use of MATLAB was effective, we started with the evaluation of the **similarity matrix** for a small subset of user (almost 9000 out of more than 135000). Even we this relatively small subset we had to spend a considerable amount of time searching for time bottlenecks with the *code-profiler* provided by the MATLAB environment. It is known that most of time problems with this programming language comes up when implementing nested for-loop structures. So, we tried to minimize the time taken by the script using array and matrix operations instead of loops, when possible.

Different tests on time performances showed real improvements when consecutive portions of the dataset were stored on a matrix: in this way we speeded up the retrieval of the list of items to pass to the method `evaluatePearson`. In the following, the pseudo-code of the method: The most important optimization step

Algorithm 1 evaluatePearson($user_1$,$user_2$)

$avg_1 \leftarrow avgItemsRateUser1$
$avg_2 \leftarrow avgItemsRateUser2$
if $(max(user_1.item) < min(user_2.item)$ || $max(user_2.item) < min(user_1.item))$ **then**
  return NaN
end if
$c \leftarrow 1$
if $(size(user_2)$ ¡ $size(user_1))$ **then**
  for i in $[1,size(user_2)]$ **do**
    if $(user_2.item(i)$ in $user_1.item)$ **then**
      $r_1(c) \leftarrow rateUser1ItemI$
      $r_2(c) \leftarrow rateUser2ItemI$
      $c \leftarrow c + 1$
    end if
  end for
else
  for i in $[1,size(user_1)]$ **do**
    if $(user_1.item(i)$ in $user_2.item)$ **then**
      $r_1(c) \leftarrow rateUser1ItemI$
      $r_2(c) \leftarrow rateUser2ItemI$
      $c \leftarrow c + 1$
    end if
  end for
end if
return $pearson(r_1,\bar{r}_1,r_2,\bar{r}_2)$

in this case is in the two if-else statements. If we consider that each item as a numeric ID, this means that if the maximum itemID rated by $user_1$ is less than the minimum rated by $user_2$ (or viceversa), there are no common items and we return NaN. Then, in order to minimize the number of comparisons we iterate on the user that reviewed less items, and we temporary save all the ratings for the items in common in two different arrays. Finally, we return the pearson coefficient obtained with Eq. 1. Other details on array manipulation can be found in the `matlab-test` folder.

After all this time-performances optimization, we encountered another problem: if we build the matrix in this way (the only one possible if we want to retrieve similarity measure in a faster way) we will end with a matrix of dimensions 135000x135000. Unfortunately, this would take with MATLAB 135.8GB of disk space, which exceeds the array size that it could manage (quoting the MATLAB compiler *"Creation of arrays greater than this limit may take a long time and cause MATLAB to become unresponsive."*).

We concluded that MATLAB wasn't suited for this type of operations and we decided to change architecture: nevertheless, given that this algorithm worked for small sets of users, we decided to use it cross-checking the coefficient values obtained with the following methods.

C++ approach

Considering the problems encountered with MATLAB, we moved the development of the recommender system in C++ environment. The steps, involved in the construction, are reported in details in the remaining of this section.

Database management

Considering the huge amount of ratings contained in the CSV file, we decided to implement *SQLite* in our architecture in order to build a database where to store our collection of values. Each row of the database's table consists of three entries: the first one contains the user's id, the second one the item's identification while the third represents the rate given to that particular item by the user; in this way we obtained a one-to-one correspondence between database and training dataset.
In order to speed up access to the database, several indexes were created. An index called *users_idx* was set up for fast retrieving of data via user ID, while another one called *items_idx* has been used the same function, this time using an item-based search. Finally we used also the index (*itemsusers_idx*), used for researches based on a double key build with user and item IDs. When dealing with big databases like this one, the creation of an index is fundamental and in our case speeded up database access of almost 100 times, comparing with no indexes case.
Moreover, considering that eq. 1 needs the average of the rates of each user, we constructed a dedicated database. Thanks to that, we don't need to recompute thousands of time the same average value. Obviously, also this database contains an index based on the user identification value.
The similarity matrix will be stored in a database formatted as follow: each row contains within the first two columns a couple of IDs that identify two users, while the third columns contains the pearson coefficient of the pair. An index between column 1 and 2 will be created; this index is really important since the database storing similarity values has a disk size of $\approx$ 60 GByte. For saving similarity values in the database, we adopted *sqlite* "BEGIN COMMIT" constructor, as reported in method *runTransaction*. Default writing operation takes long time especially considering the high dimension of similarity matrix. In this way, we can insert new values in the matrix every k values are found, therefore reducing considerably the total writing time.
The last database that we used is the one containing the predicted rating, formatted according to CSV file *"comp3208-2017-test"*.

Building similarity matrix

As explained above we started by considering user-based collaborative filtering. In order to apply this paradigm, we need to build a similarity matrix between users. The equation that gives *Paerson coefficient* for similarity is given in eq. 1.
The steps that involved the construction of the similarity matrix, exploiting *sqlite* in C++ environment, are the following:

- Given the symmetry of similarity matrix we only need to compute the lower triangular part, excluding diagonal values due to the fact that similarity between an user and itself is always 1.
- Then, as reported in the pseudocode of Algorithm 2, we split the computation of the matrix in an initialization section and two *for* cycles.
- In *initialization*, we retrieve with *query* method the database's entries of $user_1$ and $user_2$ by userID. Exploiting the aforementioned database with averages, we also retrieved the average associated to $user_1$. After that, we initialized *users* to a *dense google hash map* with hash function *pairhash*, and *avgs* to a vector that will store averages for all other users. Due to time optimization constraints, we

replace the standard c++ *unordered map* with a *dense google hash map*, that we found to be faster at looks up when using a key pair for research.

- In the first *for* cycle, we compute the first column of similarity matrix, hence similarity of $user_1$ with respect to all other users. This part of the algorithm is really important cause for every user we load from the respective databases: average of ratings and all entries of train dataset to the *dense google hash map* with key pair consisting of userID and itemID. This will reduce time computation for each of the subsequent columns.
- With the last *for* cycle we computed all other columns of the matrix. Time execution is further improved with *pragma omp tool*, thanks to which the computation of inner *for* cycle is conducted in parallel.
- At the end of every column computation, we saved only similarity values different from *NaN* in database using *runTransaction* method.
- The function *new_pearson* returns the similarity between $user_i$ and $user_j$. With reference to the code reported in Algorithm 3, we obtained an overall complexity of O(n) for Pearson's coefficient calculation, since a find operation (look up) can be easily computed in O(1) exploiting *dense google hash maps*.

Algorithm 2 similarity_matrix

$Initialization$
$user_1 \leftarrow query(user_1)$
$user_2 \leftarrow query(user_2)$
$avg_1 \leftarrow avg\_value(user_1)$
$users \leftarrow googleDenseHashMap(pairhash)$
$avgs \leftarrow vectorOfUsersAvegareges$

for i in [2,$totNumberOfUsers$] **do**
  $user_i \leftarrow query(user_i)$
  $avgs_i \leftarrow avg\_value(user_i)$
  $compare \leftarrow i$
  $sim \leftarrow new\_pearson(user_1, users_i, compare, avg_1, avgs_i)$
  if $(sim \neq 10)$ **then**
    $insertQuery.push(sim)$
  end if
end for
$runTransaction(insertQuery)$

for i in [2,$totNumberOfUsers$] **do**
  $pragma\ omp\ parallel\ for$
  for j in [$i+1$,$totNumberOfUsers$] **do**
    $compare \leftarrow j$
    $sim \leftarrow new\_pearson(user_i, users_j, compare, avg_i, avgs_j)$
    if $(sim \neq 10)$ **then**
      $insertQuery.push(sim)$
    end if
  end for
  $runTransaction(insertQuery)$
end for

Algorithm 3 new_pearson($user_i$, $user_j$, $compare$, avg_i, avg_j)

$newnum \leftarrow 0$
$newden_1 \leftarrow 0$
$newden_2 \leftarrow 0$
for it in iterator[$user_i.begin$,$user_i.end$] **do**
  $iteratorfinder = user_j.find(compare, it \rightarrow itemID)$
  if ($find \neq user_i.end$) **then**
    $diff_1 \leftarrow (it \rightarrow rating - avg_i)$;
    $diff_2 \leftarrow (finder \rightarrow rating - avg_j)$;
    $newnum \leftarrow newnum + diff_1 \times diff_2$;
    $newden_1 \leftarrow newden_1 + diff_1^2$;
    $newden_2 \leftarrow newden_2 + diff_2^2$;
  end if
end for
return

The overall computation time for the similarity matrix takes about 4, even if we manage to utilize advance data structures such as *dense google hash maps* and *pragma omp* for parallel computation.