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.

MATLAB Approach

We decided to implement *collaborative filtering* paradigm, in particular 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