Assignment 3 - Report

1. SELECTING A DATASET

In order to build a recommendations engine, I am going to be using the MovieLens 1M dataset, a stable benchmark dataset which contains 1 million ratings from 6,000 users on 4,000 movies. It was released on Feb. 2003.

2. COLLABORATIVE FILTERING RECOMMENDATION SYSTEM

Collaborative filtering is a popular recommendation algorithm that bases its predictions and recommendations on the ratings or behaviors of other users in the system. The fundamental assumption behind this method is that other users' opinions can be selected and aggregated in such a way to provide a reasonable prediction of the active user's preference. The idea is that if users agree on the quality or relevance of some items, then they will likely agree on other items as well.

I am now going to describe 3 methods for implementing a collaborative filtering recommender for the data set previously selected. More information on these methods can be found here.

Method #1 - User to User collaborative filtering

User-user collaborative filtering, also known as *k*-NN *collaborative filtering*, is the algorithmic interpretation of the core idea behind collaborative filtering: find other users whose past rating behavior is similar to the current user and then use items the user in question has not seen but has been highly regarded by the other users. It requires a rating matrix *R*, a similarity function to computer similarities among users, and a method for using similarities and ratings to generate predictions.

Computing Predictions

To generate predictions or recommendations for a user *u*, the collaborative filter first needs to find a neighborhood of users neighbors of *u*. Once the neighborhood *N* has been computed, the system combines the ratings of users in N to generate predictions for user *u*'s preference for an item *i*. This is typically done by computing the weighted average of the neighbors' ratings using similarity as the weights.

$$p_{u,i} = \bar{r}_u + \frac{\sum_{u' \in N} s(u,u')(r_{u',i} - \bar{r}_{u'})}{\sum_{u' \in N} |s(u,u')|}$$

In the previous equation, subtracting the user's mean rating r_u compensates for the differences in users' use of the rating scale (some users will tend to give higher ratings than others). The previous equation can also be extended to normalize user ratings to *z-scores* by diving the offset from the mean rating by the standard deviation of each user's ratings, compensating therefore for users differing in rating spread as well as mean spread:

$$p_{u,i} = \bar{r}_u + \sigma_u \frac{\sum_{u' \in N} s(u,u')(r_{u',i} - \bar{r}_{u'})/\sigma_{u'}}{\sum_{u' \in N} |s(u,u')|}$$

In terms of the number of neighbors to select, different possibilities exist. Some systems use all users as neighbors, others have neighborhoods for each item, etc. For movie ratings data, k = 20 seems to have worked well in other studies.

Computing User Similarity

A critical design decision in implementing user-user collaborative filtering is the choice of similarity function. Several different similarity functions have been proposed and evaluated in the literature:

- **Pearson correlation:** this method computes the statistical correlation (Pearson's r) between 2 user's common ratings to determine their similarity. The correlation is computed by:

$$s(u,v) = \frac{\sum_{i \in I_u \cap I_v}(r_{u,i} - \bar{r}_u)(r_{v,i} - \bar{r}_v)}{\sqrt{\sum_{i \in I_u \cap I_v}(r_{u,i} - \bar{r}_u)^2}\sqrt{\sum_{i \in I_u \cap I_v}(r_{v,i} - \bar{r}_v)^2}}$$

  Pearson's correlation suffers from computing high similarity between users with few ratings in common. This can be improved by setting a threshold on the number of co-rated items necessary for full agreement (correlation 1) and scaling the similarity when the number of co-rated items falls below this threshold. Experiments have shown a threshold value of 50 to be useful in improving prediction accuracy. The threshold can be applied by multiplying the resulting similarity function by min $\{|I_u, I_{u2}|/\text{threshold}, 1\}$.

- **Constrained Pearson correlation:** the idea is using the same formula as in the person correlation, but in this case, instead of using the r_u and r_v means, we use a r_z score which represents the mean value of the rating scale.

- **Cosine similarity:** in this case users are represented as vectors, and similarity is measured by the cosine distance between 2 vectors. Unknown ratings are considered 0, causing them to be negative. The formula is:

$$s(u,v) = \frac{\mathbf{r}_u \cdot \mathbf{r}_v}{\|\mathbf{r}_u\|_2 \|\mathbf{r}_v\|_2} = \frac{\sum_i r_{u,i} r_{v,i}}{\sqrt{\sum_i r_{u,i}^2}\sqrt{\sum_i r_{v,i}^2}}$$

Method #2 - Item to Item collaborative filtering

Although User-to-User collaborative filtering is effective, it suffers from scalability problems as the user base grows. Searching for the neighbors of a user is a O(|U|) operation, where U is the set of all users. *Item-to-item* collaborative filtering was developed specifically for the purpose of scalability. The idea is that if 2 items tend to have the same users like and dislike them, then they are similar and users are expected to have similar preferences for similar items. It can be seen then that in terms os structure it is similar to content based approaches to recommendation and personalization, but item similarity is deduced from user preference patterns rather than extracted from item data.

Item-item collaborative filtering generates predictions by using the user's own ratings for other items combined with those items' similarities to the target item, rather than other users' ratings and user similarities as in user-user collaborative filtering. Similar to the user-user collaborative filtering, however, the recommender system needs a similarity function and a method to generate predictions from ratings and similarities. A more detailed step by step description of the algorithm follows:

- Given an Item Matrix where every row represents an item and every attribute a user rating
- We want to generate a recommendation for user *u*
- To do this, we go over every item *i* that *u* has rated
- And find the *k* most similar items of *i* in the Matrix using a similarity function
- We then predict a rating the user will give item *i* based on the ratings he has given the similar items in the set. Since the user might not have rated all items in the set, only those he has rated are taken into account.

Just by this definition, it can be seen that the problem of scalability still exists, since since it is still necessary to find the most similar items to generate predictions and recommendations. The performance boost can be seen however when the similarity matrix is pre-computed.

In the case of User-to-User, os a user rates and re-rates items, their ratings behavior will change along with their similarity to other users. Finding similar users in advance is therefore complicated, since a user's neighborhood is determined not only by his ratings but also by the ratings of other users, so the neighborhoods can change as a result of new ratings supplied by any user in the system. For this reason, most user-user collaborative filtering systems find neighborhoods at the time when predictions or recommendations are needed.

In systems with a sufficiently high user to item ratio however, one user adding or changing his recommendation is unlikely to significantly change the similarity between 2 items, particularly when the items have many ratings. Therefore, it is reasonable to pre-compute similarities between items in an item-item similarity matrix.

Computing Predictions

After collecting a set of items S similar to *i,* the prediction $p_{u,i}$ can be computed using weighted sum as:

$$P_{u,i} = \frac{\sum_{\text{all similar items, N}} (s_{i,N} * R_{u,N})}{\sum_{\text{all similar items, N}} (|s_{i,N}|)}$$

It is worth mentioning 2 challenges that might be faced when computing predictions:

- Depending on the method used to compute similarities, these might end up being negative but ratings might be constrained to be positive. When this occurs, this can be corrected by thresholding similarities so that only those with non-negative similarities are considered or by averaging distance from the baseline predictor:

$$p_{u,i} = \frac{\sum_{j \in S} s(i,j)(r_{u,j} - b_{u,i})}{\sum_{j \in S} |s(i,j)|} + b_{u,i}$$

- When using non-real-valued rating scales, such as unary scales common in e-commerce sites, then averaging does not work: if all similarities are positive and the rating of user for an item is 1 when he has purchased it, then the prediction will always be 1. When this occurs, pseudo-predictions with a simple aggregation of the similarities to items in the user's purchasing history (using summation, for example) is a good workaround.

$$\tilde{p}_{u,i} = \sum_{j \in I_u} s(i,j)$$

Computing Item Similarity

The item-item prediction process requires an item-item similarity matrix **S**. This matrix is a standard sparse matrix, with missing values being 0 (no similarity). Cosine Similarity is the most popular and widely used metric as it is simple, fast and produces good accuracy.

$$s(i,j) = \frac{\mathbf{r}_i \cdot \mathbf{r}_j}{\|\mathbf{r}_i\|_2 \|\mathbf{r}_j\|_2}$$

Method #3 - Hybrid Recommender Systems

The use of hybrid recommender systems consist in using collaborative filtering algorithms such as user-user or item-item in combination with other approaches such as content-based approaches to consider particular use cases where the user-user or item-item tend to give worse results.

For example, item-item collaborative filtering suffers when no one has rated an item yet, but content-based approaches do not. A hybrid recommender could use description based similarities to match the new item with an existing item based on metadata, allowing it to be recommended anyway, and increase the influence of collaborative filtering as users rate the item. Similarly, users can be defined by the content of the times they like as well as the items themselves.

There have been different methods for incorporating hybrid recommenders, which include:

- *Weighted* recommenders, which take the score produced by several recommenders and combine to produce a recommendation list for the user.
- *Switching* recommenders, which switch between different algorithms and use the algorithm expected to have the best result in a particular context.
- *Mixed* recommenders, which present the result of several recommenders together. This is similar to weighting, but the results are not combined in a single list.
- *Feature-combining* recommenders, which use multiple recommendation data sources as inputs to a single meta-recommender algorithm.
- *Cascading* recommenders chain the output of one algorithm into the input of another.
- *Feature-augmenting* recommenders, which use the output of one algorithm as the input of another.

Based on this, I would like to propose a hybrid method that combines user-user, item-item, and content based approaches into a single collaborative recommendation:

- For a particular user to whom we want to provide recommendations, we find the neighborhood of the *k* users most similar to him.
- We then go over every item *i* that user *u* has rated and find the *k2* most similar items for that particular user, using as attributes only the users that are similar to *u*
- We then predict the items rating user *u* would have given item *i,* influenced by what the users most similar to u think of that particular item.

3. IMPLEMENTATION OF USER-USER COLLABORATIVE FILTERING

Even though item-item collaborative filtering is clearly superior in many instances, I decided to implement a user-user approach. The similarity function used to compare users was Pearson's correlation, and I used neighborhoods of $k = 20$ for every user.

The python script can be found on *recommender.py.*

4. EVALUATION OF RESULTS

Evaluating the results of a user-user collaborative filtering algorithm was a little trick at first, since the model is not actually trained, but generated on execution. Moreover, according to the theory of how to evaluate ranking of items seen in class, calculating the Area Under the Curve seemed to be the logical method. However, when analyzing the data in question and the best method for evaluation, in this case the AUC did not make much sense because of the continuous nature of the algorithm predictions of ratings. While it would have been possible to assign discrete values, usually the AUC is generated based on the True Positive Rate and False Positive Rate, which only work when the prediction results are binary.

Because of this, I decided to use a different evaluation mechanism for my collaborative filter: **mean absolute error**. This method simply takes the mean of the absolute difference between each prediction and rating for all held-out ratings of users in the test set. If there are n held out ratings, the MAE is calculated as:

$$\frac{1}{n}\sum_{u,i}|p_{u,i} - r_{u,i}|$$

MAE is in the same scale of the original ratings: on a 5-star scale represented as by integers [1,5], an MAE of 0.7 means that the algorithm was, on average, off by 0.7 stars. This is useful for evaluating the set in the particular context of its rating system.

In order to calculate the MAE, I need to separate the ratings of a user in training and test set, and then let the model predict a rating for every movie in the test set and compare:

- For every user, his ratings were split into a training set (50%) and test set (50%).
- The predictions for every user were then calculated based on the ratings from the training set.

- From these predictions, I gathered only the movies that existed in the test set (ratings that exist for a user but that were not used to generate the predictions).
- For every user, we now have a list of predicted movie ratings, and the list of actual ratings.
- The MAE was then calculated. **Result: 0.75**

The code for the evaluation is in *evaluation.py***.**

The result of the MAE shows how the collaborative filter I implemented was off by 0.75 stars in rating. This is not bad at all!