Bithiah Yuan
Elira Daja
DAQL SS2018

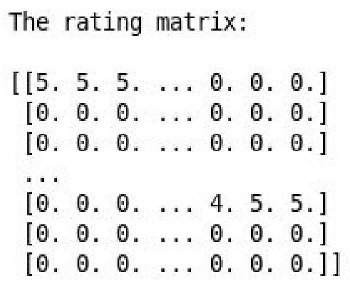
Recommender Systems Project

We implemented two recommender systems for this project and evaluated the quality of the both systems. Our recommender systems used data that contained ratings of electronic products rated by users on Amazon. Our goal was to predict a rating for an item that an active user (a user that has given a rating for an item) has not rated yet then recommend a list of items in the database for the user. We first analyzed the given data with various plots for visualization. Then we implemented the item-based collaborative filtering recommender system and an advanced recommender system that used the alternating least squares method. In the end, we implemented an evaluation system to look at the precision of each system. The Python code is in the Jupyter Notebook file where the different components of the project are available in the cells followed by example outputs. Starting from task 2 and onwards the results that are discussed are for the first 5000 rows of the data, since the computation time for the whole dataset was not achievable with the resources that we had.

Task 1 (Understand the data and add your own ratings):

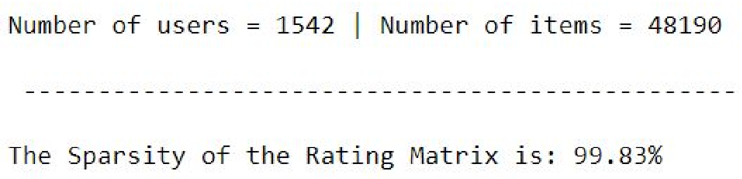
A. **Compute sparsity of ratings (in percentage %) in the utility matrix (dimensions are |Users| x |Items| and values are ratings).**

We first computed the utility matrix, which we will refer to as the rating matrix. Since the userID and itemID are stored in a dictionary, the rows of the matrix represent the indexes of the users and the columns represent indexes of the items. The rating matrix is as follows:



```
The rating matrix:

[[5. 5. 5. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 4. 5. 5.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

We counted the number of unique users and items, then computed the sparsity of the rating matrix.

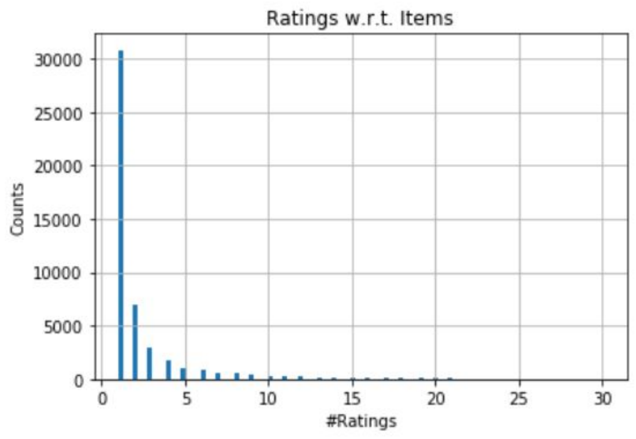


```
Number of users = 1542 | Number of items = 48190

 ------------------------------------------------

The Sparsity of the Rating Matrix is: 99.83%
```

As it is shown from the results above the sparsity of the rating matrix is 99.8%. Since sparsity is the number of the zero-valued elements divided by the total number of elements in the matrix, this means that 99.8% of the ratings in our rating matrix are 0. This means that a lot of the products have been rated only a few number of times by a few number of users. This is something normal for large datasets as the one we were given.

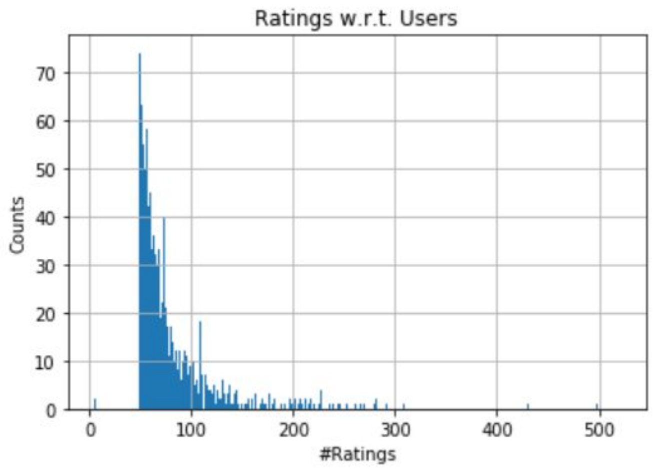
B. Do ratings follow a long-tailed distribution? Plot the histogram of ratings with respect to items to justify your answer.

We plotted the histogram of ratings with respect to items and the result is shown below in the graph. As we can see most of the products have received one rating and very few of them have received more than 5 ratings. The data follows a long tail, which again is very common with the kind of dataset that we were given.



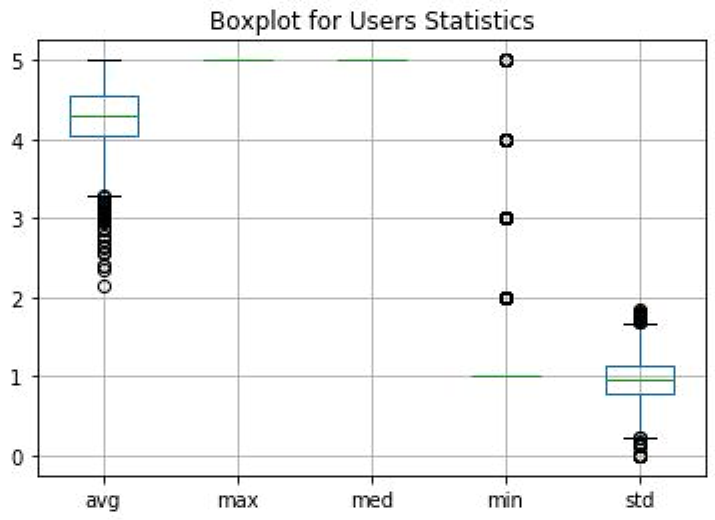
C. Plot the histogram of ratings with respect to users to get an understanding of how many ratings the majority of users tend to give and to easily identify outliers.

The plot for the ratings with respect to users is also shown below. As we can see this plot also follows a long tail. This means that the majority of users tend to give between 0 and 100 ratings and only a few users tend to give more than 100 ratings.

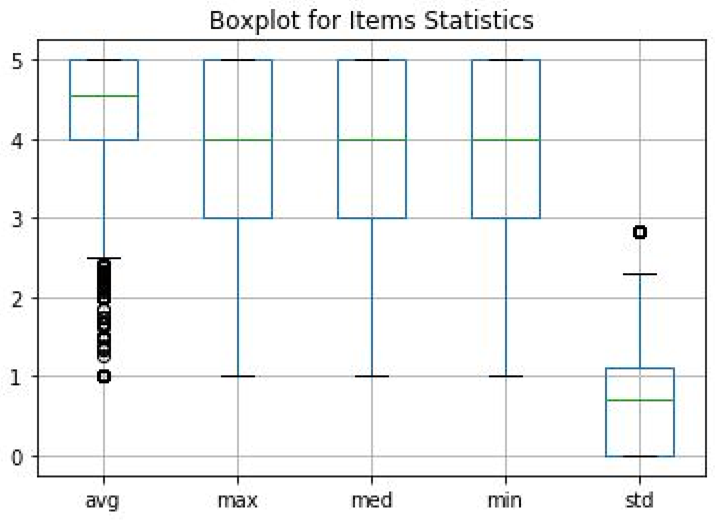


D. Calculate minimum, maximum, median, average number and standard deviation of ratings per user and per item. Show the results for both users and items in a boxplot.

The boxplots that we calculated are shown below:



For the users we see that the average rating is around 4.5, but we also have some outliers that float between 2 and 3.5. The maximum rating for almost all the users is 5 and the median as well. The min is 1, but we see that we have a few outliers of 2,3,4 and 5. The standard deviation is around 1 for most of the users, but some outliers can be seen.



From the box plot for the items' satistics, we see that the average rating of each item is 4.5 and we have outliers where the items have an average of 1 to 2.5. The maximum rating for the items is 5. The median and minimum ratings for the items are also 5. The standard deviation of the ratings for each item is around 0.75 with an outlier of about 2.8.

E. **In our dataset the rating scale is based on a 1-5 stars. Do some analysis about the usage of these rating values. Do users tend to assign high or low ratings (stars) in the dataset?**

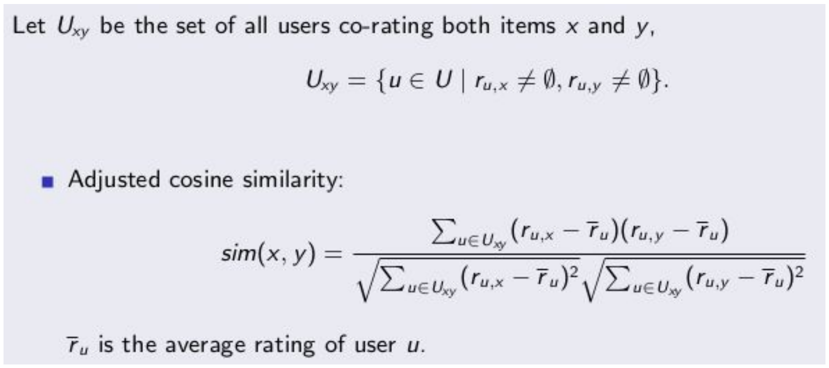
From the boxplot above we see that the average rating for the users and the items are about 4.5 so we can conclude that the users tend to give high ratings.

Task 2 (Implement a baseline Recommender System (BLRS)):

We implemented a basic Item-to-Item Collaborative Filtering Recommender System. Since user preferences remain stable and consistent over time, based on the users' previous decisions (ratings of the items that they rated), we can predict and estimate their ratings for an unrated item. It follows that we can issue an appropriate list of recommendations based on these predictions.

An item-based recommendation system provides the rating, r(a, q) of some item q for active user a. The rating is predicted based on the ratings r(a, p) assigned to items p that user a has rated which are similar to item q. Thus, we have a set L which is a set of items rated by user a that are most similar to item q. In our code we picked L = 50.

As shown in task 1, the rating matrix of the dataset has a sparsity of 99.8%. This is problematic because only a few common ratings between two items are likely. We will examine below the similarity of items by column-wise analysis of the rating matrix.


Let U_{xy} be the set of all users co-rating both items x and y,

$$U_{xy} = \{u \in U \mid r_{u,x} \neq \emptyset, r_{u,y} \neq \emptyset\}.$$

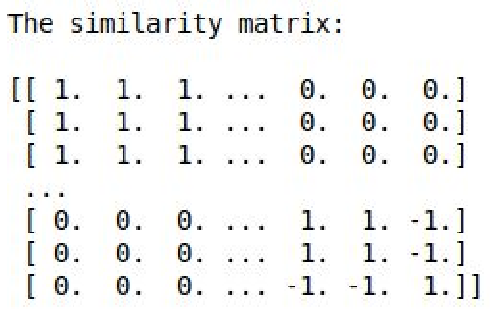
- Adjusted cosine similarity:

$$sim(x, y) = \frac{\sum_{u \in U_{xy}} (r_{u,x} - \overline{r}_u)(r_{u,y} - \overline{r}_u)}{\sqrt{\sum_{u \in U_{xy}} (r_{u,x} - \overline{r}_u)^2} \sqrt{\sum_{u \in U_{xy}} (r_{u,y} - \overline{r}_u)^2}}$$

$\overline{r}_u$ is the average rating of user u.

For faster run-time, we made the computations and example outputs for the first 5000 rows of the data. See the function *simM* in the class *BLRS( )* for the computation of the adjusted cosine similarity matrix. We first computed the mean-adjusted rating matrix by computing the average rating per user by taking the sum of ratings of each user which is stored in the variable *tot*. This is shown in the function *avgR.* We made sure we don't consider unknown ratings which are converted to zero in the code. We store the total number of non-zero ratings in the variable *cts*. Then we computed the average rating per user by dividing *tot* by *cts*. The returned variable *mu* is an array of average ratings per user, where the index corresponds to the user index in the rating matrix.

Then in the *simM*, we created a matrix with *mu*. We made sure when we subtract the rating matrix by the matrix of mu, that the unknown ratings which are represented as zeros are not subtracted (it doesn't become a negative number, but remains zero instead). This is seen in the construction of the mean-adjusted matrix. Then we used a built-in function to calculate the cosine similarity of the items which goes through all the columns of the mean-adjusted matrix which represent the different items and takes the dot product divided by the product of the norm of a pair of items. Different from the formula given in the lecture, the built-in function makes sure that even if two items don't share a single rating, the ratings of that item is still included in the calculation of the norm. We did this to avoid getting too many cosine similarities that are 1. The resulting similarity matrix (*sim_matrix)* is as follows:

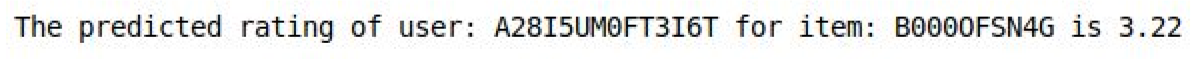


```
The similarity matrix:

[[ 1.  1.  1. ...  0.  0.  0.]
 [ 1.  1.  1. ...  0.  0.  0.]
 [ 1.  1.  1. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ...  1.  1. -1.]
 [ 0.  0.  0. ...  1.  1. -1.]
 [ 0.  0.  0. ... -1. -1.  1.]]
```

We then implemented the prediction function *predictRating* following the equation:

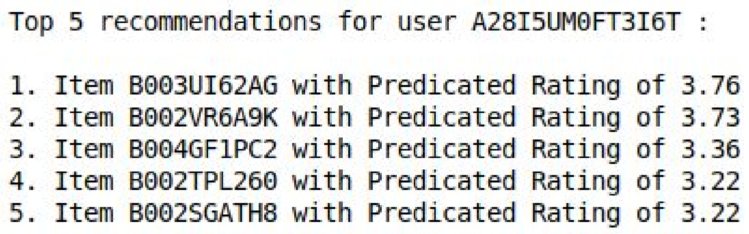
$$pred(a, q) = \frac{\sum_{p \in L} sim(q, p) \cdot r(a, p)}{\sum_{p \in L} \mid sim(q, p) \mid}$$

where L denotes the set of items rated by active user a that are most similar to item q. This method returns for a given user (the active user) and item a predicted rating. After getting the user and item index from the dictionary with the userID and itemID, the function checks the rating matrix to see if the user has given a rating for the item. If not, it looks for the column in the *sim_matrix* corresponding to the item index which is sim(q,p) stored in the array *v*. Then it stores the column in an array and disregards the similarities that are negative to get a better prediction. It sorts the array and picks the top 50 similar items (L = 50). The index of the sorted similarities are stored in the array *item_num*. Then it goes to the rating matrix to find the given user's ratings corresponding to the 50 most similar items which is r(a, p) which is stored in the array *item_rating*. We then take the dot product of *v* and *item_rating* and the sum of *v* (all the similarities). We finally divide the dot product by the similarity sum to get the predicted rating of the given item. A sample output:



```
The predicted rating of user: A28I5UM0FT3I6T for item: B000OFSN4G is 3.22
```

With the concept of the above function, we implemented the function *predictTopKRecommendations*, which returns a list of top k recommendations for a given active user. This function has the same logic as the *predictRating* function. Except we only give the argument of a given user. We go to the rating matrix to look for the unrated items so we can get the item similarities for each unrated item. Then the function calculates the predicted rating for the unrated items and sorts the ratings to return the top k recommendations. A sample output:



```
Top 5 recommendations for user A28I5UM0FT3I6T :

1. Item B003UI62AG with Predicated Rating of 3.76
2. Item B002VR6A9K with Predicated Rating of 3.73
3. Item B004GF1PC2 with Predicated Rating of 3.36
4. Item B002TPL260 with Predicated Rating of 3.22
5. Item B002SGATH8 with Predicated Rating of 3.22
```

Task 3 (Implement an advanced Recommender System (ARS)):

For task 3 we implemented an advanced recommender system. We chose to implement a recommender system that uses the ALS(Alternating Least Squares) algorithm with U-V factorization following the steps of the algorithm presented in the tutorial and the source that was given in the project specifications:
http://activisiongamescience.github.io/2016/01/11/Implicit-Recommender-Systems-Biased-Matrix-Factorization.
We implemented the basic version of the algorithm without the user and item biases or any other regulations.
The idea is to find the parameters x_u^j and y_i^j (the entries of the matrices X and Y, which correspond to Users and Items in our case) which minimize the L^2 cost function:

$$C = \sum_{u,i \in \text{observed ratings}} (r_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

So in order to minimize C, one could try the following:

1. Hold the user vectors fixed and *solve* the quadratic equation for the y_i^j's. This will (probably) not be the global minimum of C since we haven't touched half of the variables (the x_u^j 's), but we have at least decreased C.
2. Hold the item vectors fixed and *solve* the quadratic equation for the x_u^j's.
3. Repeat.

This logic gives us the following steps of the ALS algorithm which we implemented in the code:

1. Initialize the user vectors X somehow (e.g. randomly).

2. For each item i, let r_i be the vector of ratings of that item (it will have n_users components; one for each user). Compute:

$$y_i = \left(X^T X + \lambda I\right)^{-1} X^T r_i$$

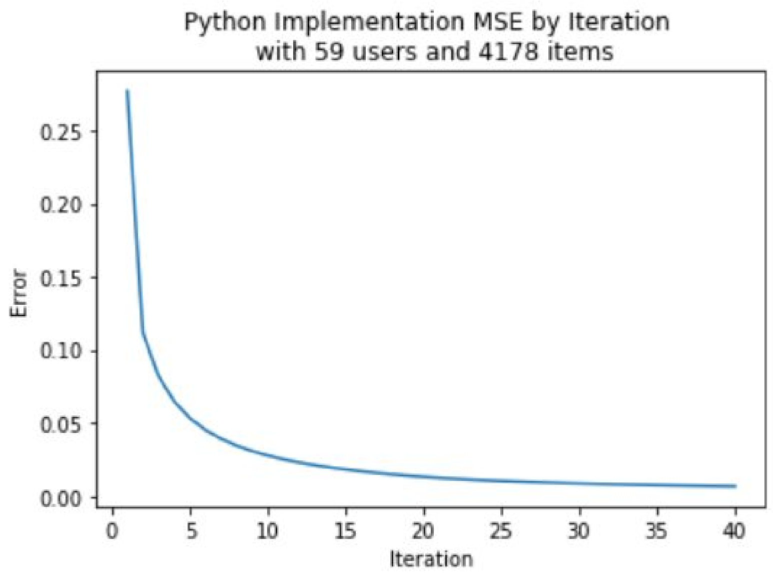
   for each item i. (I is the NxN identity matrix.).
3. For each user u, let r_u be the vector of ratings of that user (it will have n_items components; one for each item). Compute:

$$x_u = \left(Y^T Y + \lambda I\right)^{-1} Y^T r_u$$

   for each user u.

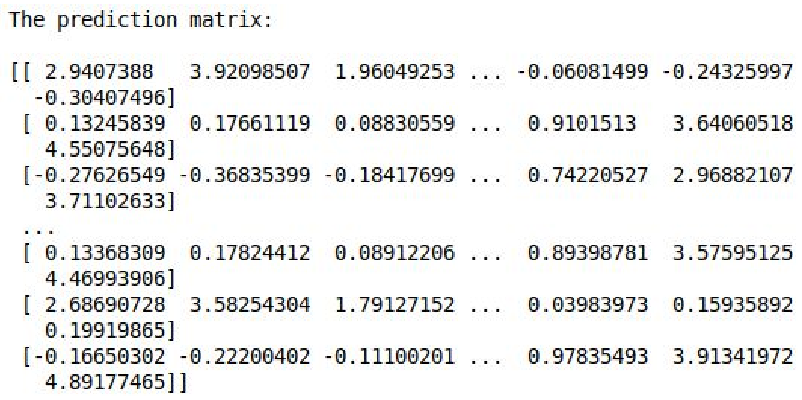
4. Repeat 2), 3) until desired level of convergence is achieved.

In the code we also calculated the MSE (mean squared error) so that we have a better understanding of what parameter values give the least error so that we know when the convergence is achieved. After 20 iterations we noticed that the error starts decreasing slower and the difference is not that visible anymore. So to save time we decided on the number of iterations 20.

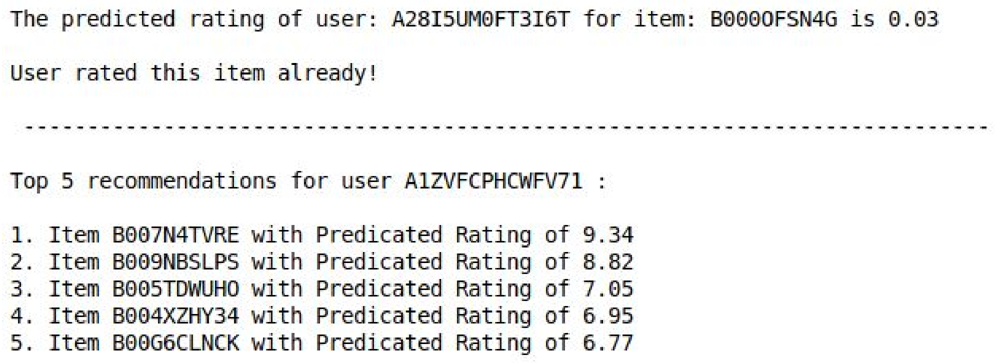


The above plot is for 40 iterations and shows what we already mentioned above.

After running the algorithm the prediction matrix looks like this:


```
The prediction matrix:

[[ 2.9407388   3.92098507  1.96049253 ... -0.06081499 -0.24325997
  -0.30407496]
 [ 0.13245839  0.17661119  0.08830559 ...  0.9101513   3.64060518
   4.55075648]
 [-0.27626549 -0.36835399 -0.18417699 ...  0.74220527  2.96882107
   3.71102633]
 ...
 [ 0.13368309  0.17824412  0.08912206 ...  0.89398781  3.57595125
   4.46993906]
 [ 2.68690728  3.58254304  1.79127152 ...  0.03983973  0.15935892
   0.19919865]
 [-0.16650302 -0.22200402 -0.11100201 ...  0.97835493  3.91341972
   4.89177465]]
```

It is not visible in the matrix above but we get predictions which are more than 5. This in our case does not make sense cause the ratings are from 1 to 5. This means that the basic ALS algorithm does not work well and does not give good results for rating matrices that are so sparse. The results below show the same.


```
The predicted rating of user: A28I5UM0FT3I6T for item: B000OFSN4G is 0.03

User rated this item already!

 -------------------------------------------------------------------------

Top 5 recommendations for user A1ZVFCPHCWFV71 :

1. Item B007N4TVRE with Predicated Rating of 9.34
2. Item B009NBSLPS with Predicated Rating of 8.82
3. Item B005TDWUHO with Predicated Rating of 7.05
4. Item B004XZHY34 with Predicated Rating of 6.95
5. Item B00G6CLNCK with Predicated Rating of 6.77
```

As shown, we get top ratings which are more than 5, which in turn makes the number of relevant items be the maximum of items and gives us evaluations which are not believable, but we will also explain this in the evaluation task to follow.

Task 5: Implement an Evaluation System for Recommender Systems

For the evaluation system, we first combined folds 1 to 4 for the dataset to be our training set. Then we loaded fold 5 as the test set. We found the common users in both the training and test set by concatenating the column of userIDs in both files and look for duplicates by the built-in function *duplicated*. From these common users, we then chose 10 random users. We return the top 10 recommendations for each of the 10 users for the evaluation of our recommender systems. For faster run-time, we made the computations and example outputs for the first 2000 rows of the data.

In both the *BLRS()* and *ALS()* class, we have the *revItems* or *revItemsALS* function both with the same logic. These function finds the relevant items for each user by applying the binary judgement where if the predicted rating of a user is greater than the average rating of the user, it is considered as a relevant item for the user. Thus, the function calls *avgR* to find the average rating of the given user (supplied in the array of 10 random users) and calls the *predictTopKRecommendations* function to find the top 10 item ratings. It compares the item ratings with the average and stores the relevant items in an array *rev_items*. It gets the number of relevant items and stores the value in *num_rev*. To calculate the precision of the recommender systems, we used the following formula:

$$\text{Precision@M} = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \frac{\#\ \text{recommended items @M that are relevant to } u}{M}$$

We calculated Precision@10 for the top 10 items relevant to the 10 random users. In the *precision* or *precisionALS* function, we called the *revItems* or *revItemsALS* function to get the number of recommended items @10 that are relevant to the users. We divided this number by 10 for each user and summed the results and multiplied by $\frac{1}{10}$ where 10 is the number of random users to obtain the precision.

For BLRS, our precision is 0. This is likely as a result of the sparsity of the matrix.
For ALS, our precision is 1. This is because often the predicted ratings are above 5. Thus, it will be greater than the average rating of the user and be considered as a relevant item.

In conclusion, BLRS didn't achieve good results due to the lack of ratings by an active user to be able to compute accurate item similarities. Whereas, ALS overpredicted and returned inappropriate ratings.