Neighborhood based collaborative filtering (NBCF)

~Project report~

Yaser Z. K. Shoshaa 2104449

Firas Elbayoumi 2100503

Briksam Kasimoglu 2102969

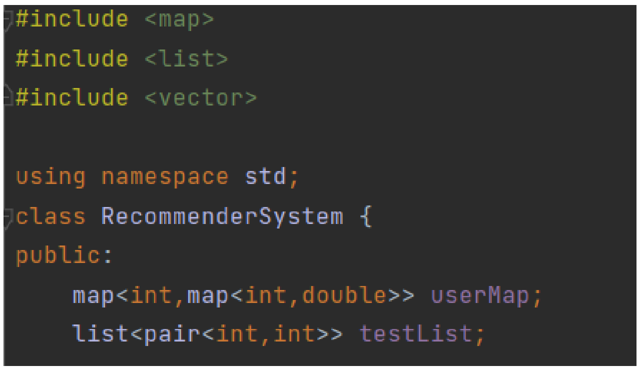
1. Introduction

We have approached the recommender system problem that requires us to find most rated items and users with the most ratings afterwards we must predict missing items rating using neighborhood based collaborative filtering (NBCF) algorithm while using RMSE to measure error.

2. Data structures

After looking at the provided train and test files with research we have decided to use maps due to easy item accessibility and quick navigation another reason is the low time complexity and overall efficiency with large amount of data

Maps work as the following we have unique keys and corresponding values for instance a real-life example is students grades with keys being student ids/names and the corresponding value as grade.



```
#include <map>
#include <list>
#include <vector>

using namespace std;
class RecommenderSystem {
public:
    map<int,map<int,double>> userMap;
    list<pair<int,int>> testList;
```

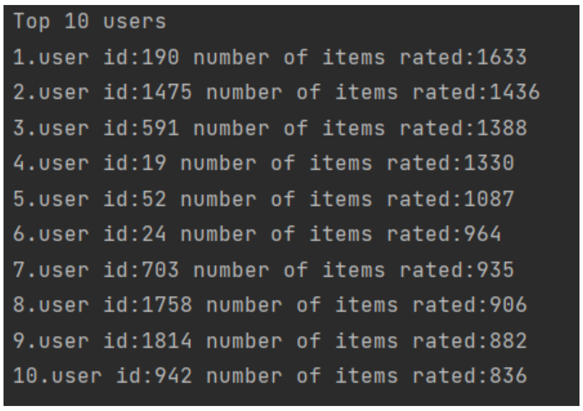
In our case we have two main maps RS and RS_rev with the 2nd one being a reversed version of the first one. RS has the following structure:

User id (key for outer map): Item id (value for outer map as well as a key for inner map): Rating (value for inner map)

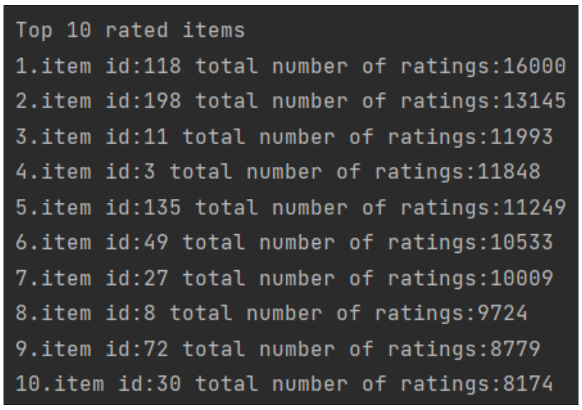
For RS_rev we have the following structure:

Item id (key for outer map): User id (value for outer map as well as a key for inner map): Rating (value for inner map)

3. Top users and items



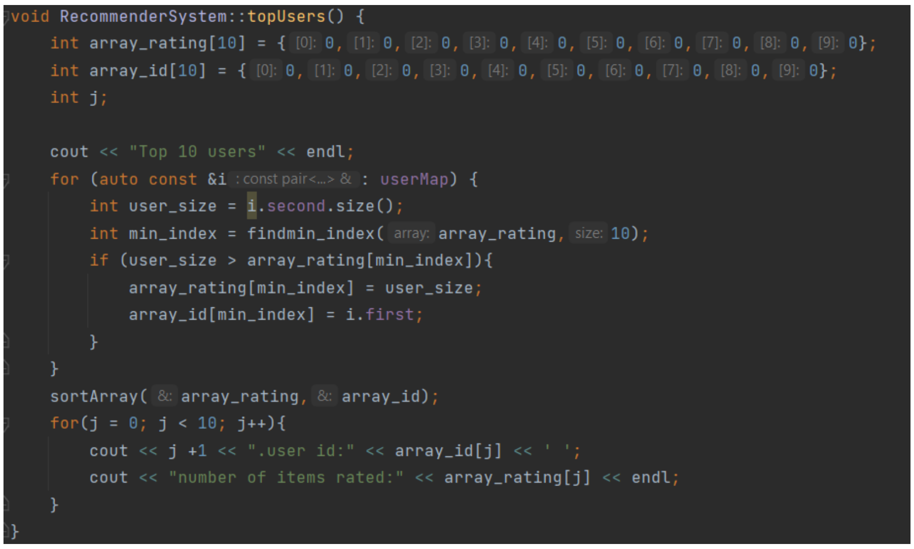
```
Top 10 users
1.user id:190 number of items rated:1633
2.user id:1475 number of items rated:1436
3.user id:591 number of items rated:1388
4.user id:19 number of items rated:1330
5.user id:52 number of items rated:1087
6.user id:24 number of items rated:964
7.user id:703 number of items rated:935
8.user id:1758 number of items rated:906
9.user id:1814 number of items rated:882
10.user id:942 number of items rated:836
```



```
Top 10 rated items
1.item id:118 total number of ratings:16000
2.item id:198 total number of ratings:13145
3.item id:11 total number of ratings:11993
4.item id:3 total number of ratings:11848
5.item id:135 total number of ratings:11249
6.item id:49 total number of ratings:10533
7.item id:27 total number of ratings:10009
8.item id:8 total number of ratings:9724
9.item id:72 total number of ratings:8779
10.item id:30 total number of ratings:8174
```

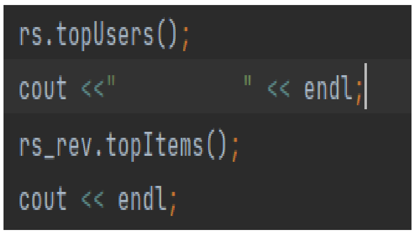
we made a function to obtain top ten users firstly we created two empty arrays with the size of 10 to store the user id with greatest number of ratings in the first array and store the number of rated items by that user in the second array to do so we loop over the outer map (RS) if the current user has more number of ratings then the lowest user in the array we replace it as well as the corresponding value in the second array until we loop over the whole map.

For top ten rated items we made the same function however instead of applying it to RS we used the reversed version RS_rev.



```
void RecommenderSystem::topUsers() {
    int array_rating[10] = { [0]: 0, [1]: 0, [2]: 0, [3]: 0, [4]: 0, [5]: 0, [6]: 0, [7]: 0, [8]: 0, [9]: 0};
    int array_id[10] = { [0]: 0, [1]: 0, [2]: 0, [3]: 0, [4]: 0, [5]: 0, [6]: 0, [7]: 0, [8]: 0, [9]: 0};
    int j;

    cout << "Top 10 users" << endl;
    for (auto const &i : const pair<...> & : userMap) {
        int user_size = i.second.size();
        int min_index = findmin_index( array: array_rating, size: 10);
        if (user_size > array_rating[min_index]){
            array_rating[min_index] = user_size;
            array_id[min_index] = i.first;
        }
    }
    sortArray( &: array_rating, &: array_id);
    for(j = 0; j < 10; j++){
        cout << j +1 << ".user id:" << array_id[j] << ' ';
        cout << "number of items rated:" << array_rating[j] << endl;
    }
}
```

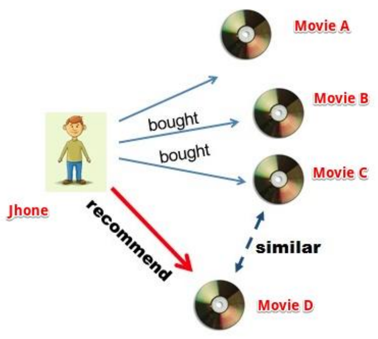


```
rs.topUsers();
cout <<"            " << endl;
rs_rev.topItems();
cout << endl;
```

4. NBCF

We have decided to implement IBCF instead of UBCF after doing some research and trying both since many resources and our own testing proved that IBCF provides lower RMSE as well as higher precision of the results.

Item based collaborative filtering:
Is an algorithm that recommends an item to a user based on how similar it is to the other items that the user prefers.



Finding out the most similar items:

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}}$$

Where,

r = Pearson Correlation Coefficient

x_i = x variable samples y_i = y variable sample

$\bar{x}$ = mean of values in x variable $\bar{y}$ =mean of values in y variable

In our implementation we applied Pearson's correlation between two vectors consisting of items rated by X user and items rated by Y user
To calculate the similarity between them, after some testing we found out that using the top 20 similar neighbors gives us the lowest RMSE scores.

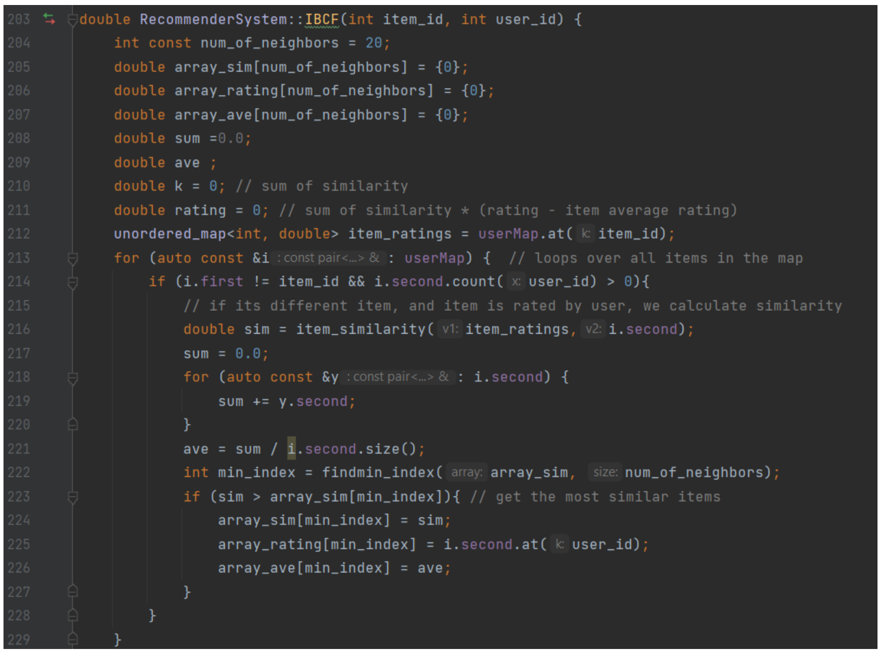
$$score(u,i) = \frac{\sum_j^I similarity(i,j)(r_{(u,j)} - \bar{r_j})}{\sum_j^I similarity(i,j)} + \bar{r_i}$$

u = user for which we are generating recommendation

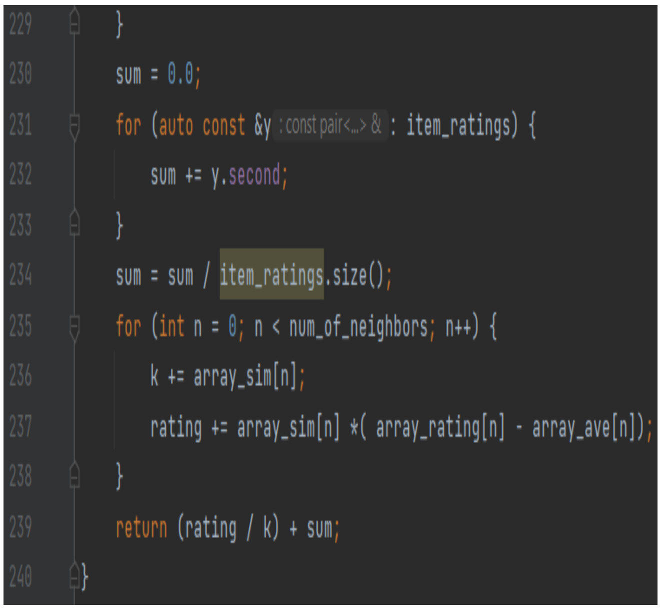
i = item in consideration, i.e. if this item should be recommended or not

score(u,i) = generates a score that will indicate how strong a recommendation of item 'i' is to our user 'u'.

j = items similar to main item i. The above equation elicits that in order to calculate recommendation score of an item ‘i’ for a user ‘u’ sum the multiplication of an item ‘i’ and ‘j’ similarity with the difference of rating given by user ‘u’ to an item ‘j’ and the average rating of an item ‘j’. Divide the result with the sum of item ‘i’ and ‘j’s similarity, add the output with the user ‘i’s average rating.



```cpp
double RecommenderSystem::IBCF(int item_id, int user_id) {
    int const num_of_neighbors = 20;
    double array_sim[num_of_neighbors] = {0};
    double array_rating[num_of_neighbors] = {0};
    double array_ave[num_of_neighbors] = {0};
    double sum =0.0;
    double ave ;
    double k = 0; // sum of similarity
    double rating = 0; // sum of similarity * (rating - item average rating)
    unordered_map<int, double> item_ratings = userMap.at( item_id);
    for (auto const &i : userMap) {  // loops over all items in the map
        if (i.first != item_id && i.second.count( user_id) > 0){
            // if its different item, and item is rated by user, we calculate similarity
            double sim = item_similarity( item_ratings, i.second);
            sum = 0.0;
            for (auto const &y : i.second) {
                sum += y.second;
            }
            ave = sum / i.second.size();
            int min_index = findmin_index( array_sim,  num_of_neighbors);
            if (sim > array_sim[min_index]){ // get the most similar items
                array_sim[min_index] = sim;
                array_rating[min_index] = i.second.at( user_id);
                array_ave[min_index] = ave;
            }
        }
    }
```



```cpp
    }
    sum = 0.0;
    for (auto const &y : item_ratings) {
        sum += y.second;
    }
    sum = sum / item_ratings.size();
    for (int n = 0; n < num_of_neighbors; n++) {
        k += array_sim[n];
        rating += array_sim[n] *( array_rating[n] - array_ave[n]);
    }
    return (rating / k) + sum;
}
```

First, we loop over the map if the current item is different than the item we want to rate, and it is also rated by the same user we calculate the similarity then the average of the current item is calculated for later use. For obtaining the top 20 similar neighbours we check whether the current item similarity is higher than the least similar item in array_sim and replace it if the condition is satisfied.

Later on, we calculate the average of item ratings then we use the equation mentioned above score (u,i).

5. Conclusion

Finally, we got RMSE score of 0.91923 on the Kaggle leaderboard taking the number 1 spot under team name “Yaser, Briksam, Firas” our code has a total runtime of 11.5 mins, All of us have been meeting 2-3 times a week at least with each meeting being 4-5 hours consisting of researching, coding and debugging collaboratively.