CS-449 Project Milestone 1: Personalized Recommender with k-NN

Name: Ana-Arina Raileanu
Sciper: 337628
Email: ana-arina.raileanu@epfl.ch

Name: Zhecho Mitev
Sciper: 323387
Email: zhecho.mitev@epfl.ch

March 24, 2022

1 Introduction

With the goal of predicting movie ratings, we implemented all suggested prediction models, whose results we will further discuss. We spent a significant amount of time improving the performance of the k-NN model. As such, an important part of our analysis will be dedicated on the changes we implemented to improve its running time.
Implementation-wise, we grouped the models of each section of the milestone (e.g. Baseline, Personalized) [1] into classes. As one model builds on top of the other, with many of the methods overlapping, we use inheritance to reduce the amount of duplicate code.

2 Baseline: Prediction based on Global Average Deviation

2.1 Questions

B.1 Table 1 shows the results that we obtain for each computation. In order to compute the global average rating we use the 'mean' helper function provided in the code. For the user average and the item average we obtain the results in two different ways to check correctness. One is directly filtering the ratings of user 1 or item 1 and then computing the average. The second approach computes the averages, given a training set, for all possible users or items and stores it in a Map. Note that this solution is much slower than the first one, however for the next questions storing the values will help us improve the execution time when we have to compute the predictions multiple times. We also note that the global average rating is quite close to the average for user 1, so it might be indeed a good idea to use the global average as an approximation of the user average when a user is missing.

Task	Value
$\bar{r}_{\bullet,\bullet}$	3.5264
$\bar{r}_{1,\bullet}$	3.6330
$\bar{r}_{\bullet,1}$	3.8882
$\bar{\hat{r}}_{\bullet,1}$	0.3027
$p_{1,1}$	4.0468

Table 1: Results for B.1.

B.2 On Table 2 we can observe that the global average predictor outputs the highest error, while the user average and the item average manage to improve the errors. We note that the baseline formula (Equation 5 [1]) outperforms the other predictors, as expected.

Task	Value
Global Average MAE	0.9489
User Average MAE	0.8383
Item Average MAE	0.8207
Baseline MAE	0.7604

Table 2: Results for B.2.

B.3 In order to compute the execution time of each method we create a Map which stores the name of the predictor as a key and sequence of 3 time measurements. We every time we make a measurement we initiate a new class 'BaselineSolver' in order to make sure everything the code is running from the start and there are no precomputed values. Figure 1 is a bar chart which shows the execution time in seconds and the MAE for each predictor. Additionally we include error bars to show the standard deviation which has been recorded during experiments. We run our experiments on a gaming laptop with the following specifications:

– Processor: Intel Core i7-8750H CPU 2.20GHz

– RAM: 16.0 GB

– OS: Windows 10

– Scala version: 2.11.12

– JVM: Oracle 1.8.0_201

We note that the global average is computed approximately in 3 milliseconds, while the user average is 3 times slower (9 ms). The item average MAE takes 18 ms to be computed, which is due to the fact the total users are 943 while the total items are 1648. Since there are more items than users, repeating the average computation per item is more costly. Thus, it seems that using the average predictors has a linear increase in time with respect to the amount of averages that need to be computed.

The baseline predictor is the most expensive to compute - it runs much slower compared to the previous predictors as we need to normalise the whole dataset first and then compute the average deviation. In our initial implementation we normalise every single `(user, item, rating)` pair, which introduces overhead. Potentially, in order to improve this time we can use a Map to store the global average deviation for each item when encountered in the dataset. We note that the baseline is the best solution in terms of MAE, however we sacrifice time as it grows with the decrease of error according to figure 1.
We notice a relationship between MAE and time that could be exponentially decreasing. By drawing a straight line between the means of the baseline and the global predictors, the user and the item average are below it, which is not characteristic of a linear decrease. This observation also depends on the implementation of the predictors.
Therefore, we conclude that accurate predictions require a significant amount of time and for larger datasets we should try to improve the method for computing predictions in order to make it scalable.

3 Spark Distribution Overhead

3.1 Questions

D.1 Due to the fact that the 25m dataset is 250 times larger than the previous one, we would like to compute the required values as fast as possible, without using data structures and storing information. Therefore, we use standard functional programming to compute the values with RDDs. Since we want to make the functions generic and test their performance for the computation of the MAE, we utilize the signature `(Int, Int) => Double` to generate a prediction for a single pair `(user, item)`. We place this function inside a closure that receives an RDD as input, similarly to the predictors in the Baseline implementation. Furthermore, we create a function `getSingleBaselinePrediction`, which computes the user average for a given user and the item global average deviation for a given item, instead of all of them. Thus the computation is very quick for as single prediction, as it allows us to avoid computing the averages of all users and items. The results we obtain with the distributed predictor are consistent with the results of the baseline.

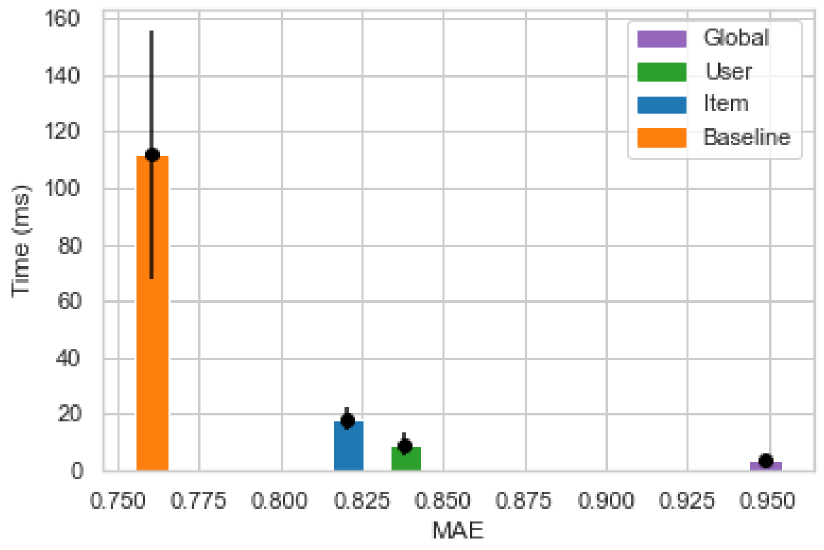

Figure 1: Bar plot of MAE of 4 different predictors

D.2 Initially, we have approached this task directly by trying to use the `getSingleBaselinePrediction` function for the whole dataset. However, such computation takes a lot of time, due to the fact that user average and item average deviations are computed for all 25 million entries. Through the closure function, which takes the training set as a parameter, we precompute a `Map[Int, Double]` for the user averages and a `Map[Int, Double]` for the item deviations. Thus we compute them only once during the whole execution and significantly reduce the execution time.

In Table 3 we can observe that the average time for running the distributed baseline solution is around 28 seconds if 4 workers are employed and 56 seconds if there is only 1 worker. The execution is two times and not four times faster due to the overhead that is introduced by the RDDs. We estimate that for around 18 seconds there is no distribution of the computation. This means that in the solution with 4 executors they only work in parallel for around 10 seconds. Thus the portion of linear work takes longer than the distributed one.

Even though our baseline implementation was already faster than the one which uses Spark, we aim to make it even better. As such, we replaced our data structures with mutable Maps and Arrays, and we delayed computation of averages until needed. We created a function `getAvgRatingByUser` which we use to obtain the average of a user. If it was already computed, it returns the value stored in the mutable Map. Otherwise, it computes the user average for a single user. Furthermore, when we initialize the `BaselineSolver` class, we group all of the ratings by user and we store them, so that at a later time we would not need to execute the costly group operation. With these improvements, we managed to achieve a running time of about 6 seconds, which is significantly lower than the distributed implementation.

	Baseline	Optimized Baseline	1 worker	4 workers
Time (s)	19.5899	6.5005	56.6326	27.9286
StdDev (s)	3.7835	0.2881	1.9975	1.6505

Table 3: Time comparison between baseline and distributed implementations.

4 *Personalized* Predictions

For the implementation of personalized predictions, we decided to use Eq. (9) and (10), as described in Section 5.2 of Milestone 1[1], to speed up our implementation.

4.1 Questions

P.1 Table 4 presents the scores for the uniform predictor, where similarity between any two users was considered to be 1.

P.2 The personalized predictor with the adjusted cosine similarity as a measure of user similarity has a lower MAE than the original predictor. The results of this model can be seen in table 5.

P.3 In order to compute the Jaccard Coefficient, we applied the formula provided on the Wikipedia page [1]. We count the number of items that both user rated and we divide it by the number of unique items rated by at least one of them. As such, given ratings, we computed the similarity according to Eq. (1).

$$s_{u,v} = \begin{cases} \frac{|I(u) \cap I(v)|}{|I(u) \cup I(v)|} & (I(u) \cup I(v)) \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Table 6 presents the results for the Jaccard Coefficient. The MAE score is worse than that of the predictor using the Adjusted Cosine similarity. We believe one possible explanation for this observation is that the Jaccard Coefficient only looks at the proportion of items both users rated. On the other hand, the Adjusted Cosine similarity uses more context about the users, as it takes into account the actual ratings each user gave to the items in common.

[1]https://en.wikipedia.org/wiki/Jaccard_index

Task	Value
Prediction User 1 Item 1	4.0468
MAE	0.7604

Table 4: Metrics for personalized prediction with *Uniform* similarities.

Task	Value
Similarity User 1 - User 2	0.0730
Prediction User 1 Item 1	4.0870
MAE	0.7372

Table 5: Metrics for personalized prediction with *Adjusted Cosine* similarities.

5 Neighbourhood-Based Predictions

For the k-NN predictor, we tried to reuse as much as possible of the code previously written. Therefore, the implementation uses methods defined for the previous tasks, and overrides the computation of the predicted score by avoiding to consider ratings of user that are not among the k closest neighbors, based on the Adjusted Cosine similarity.

5.1 Questions

5.1.1 N.1

Considering $k = 10$, the results of the nearest-neighbour predictor are presented in table 7.

5.1.2 N.2

We tested various values of k and we show the error as measured by MAE in table 8. The lowest k for which the MAE is better than that of the baseline predictor is 100. Therefore, we would need to use k-NN with 100 neighbours to outperform it.
Based on this analysis, it seems that the best score is obtained when the number of neighbours is 943, where we obtain the same MAE score as for the Personalized predictor. It seems that with low values of k the model is overfitting, and increasing that number improves its generalization.

5.1.3 N.3

After implementing all of the tasks, our model with $k = 300$ ran in 98496.4487 ms. By refactoring and comparing different strategies, we managed to reduce the time to 2321.8027 ms. Below, we discuss some of the changes we made and their impact on time. We will also provide the code before and after some of the changes.

Task	Value
Similarity User 1 - User 2	0.0318
Prediction User 1 Item 1	4.0982
MAE	0.7556

Table 6: Metrics for personalized prediction with *Jaccard* similarities.

Adjusted Cosine Similarity Initially, in order to compute the Adjusted Cosine Similarity, we would combine the lists of normalized ratings of two users, group them by item, keep the groups that had exactly two elements (i.e. they were rated by both users) and perform another map to extract the two-element array. Afterwards, we would multiply the ratings in the array of two-element arrays through another map 1. Since this approach requires iterating through all of the values of the map multiple times, it is time-consuming. Therefore, we refactored it to remove the costly functional calls. Listing 2 portrays our second implementation, which iterated through all of the normalized ratings of the two users, and if they refer to the same item, multiply them and add them to the similarity score. Even though the second implementation is not the most efficient one, it allowed us to lower the time to 37673.5687 ms, which is three times less than the initial one.

Listing 1: Adjusted Cosine Similarity with groupBy, filter, map

```
val intersection = (user1Ratings ++ user2Ratings).groupBy(x => x.item)
                                          .filter {case (_, v) =>
                                              v.length == 2}
                                          .map { case (_, v) => v }
intersection.map(x => x.head.rating * x(1).rating).sum
```

Listing 2: Adjusted Cosine Similarity with iteration

```
var similarity = 0.0
for (a <- user1Ratings; b <- user2Ratings) {
    if (a.item == b.item) {
        similarity += a.rating * b.rating
    }
}
```

Delayed Computation One change that helped us significantly decrease the time for k-NN was to delay the computation of certain values until they were needed for prediction, and then store the computed values for later use. For example, instead of pre-computing the average per user for all users in the training stage, we compute the average rating of a user only when needed for a prediction. We can apply this idea of ”delayed computation” on multiple functions:

- **Preprocessing Denominator** When computing the Adjusted Cosine

Similarity of two users, we preprocess the ratings according to Eq. 9 of the Milestone 1 [1] statement. We observe that for every item i belonging to a user u, the denominator of the preprocessed rating $\breve{r}_{u,i}$ has the same formula, as it depends on all items rated by the user. Therefore, our second approach was to compute them once for all users of the dataset, as can be seen in listing 3. A further improvement is to not recompute all denominators of all users of the training set, but instead delay this computation until the first occurrence of a user in the test dataset, and only compute it for that user. Afterwards, we store the value of the denominator, in case we have multiple ratings in the test dataset from the same user. This approach can be seen in listing 4.

- **Average User Rating** For a given user, we compute and store the average of their rating only after they are first needed in the test dataset.
- **Normalized Ratings** We normalize a set of ratings of a given user/item only when first needed. After normalizing them, we store the normalized ratings in two dictionaries. One of them allows us to access all of the normalized ratings related to an item, the other one lets us select ratings by user.
- **User Similarity** At first, we pre-computed the similarities between any two users. Since it is possible that some users will not appear in the test set, we may not care about their similarity with all of the other users. In that case, we compute the similarity between a user and the others only when we need to find which are their k-nearest neighbours.
- **k-Nearest Neighbours** Likewise, we do not aim to find the k-nearest neighbours of a user until they appear in the test set.

By using all of the techniques mentioned above, we reduce the running time to 13106.2385 ms.

Finding the k-Nearest Neighbours Initially, when we computed the k-nearest neighbours, we used `take(k)` and then `map(x => x._1)` on the list of sorted tuples `(user, similarity)` to get the list of the k-nearest neighbours. We later changed our approach to iterate the list of sorted similarities in a `for` loop and stop once we found k users. Furthermore, we store both the users and their similarity in a dictionary. By doing so, we reduced the time to run to 7470.7791 ms.

Further Improving the Adjusted Cosine Similarity The Adjusted Cosine Similarity seemed to be a bottleneck of our model. Therefore, we spent some time trying to make it as efficient as possible. The final implementation is more lengthy, but it helped us achieve a time of 5513.0615 ms. The idea behind is that instead of iterating through all combinations of ratings between the first user and the second one, as we did in listing 2, we get the normalized ratings of each user in the form of a dictionary of items to ratings. Afterwards, we check which of the users has fewer items, and we iterate through their list of items. We then lookup for the item in the other user's dictionary, and if we find it, we add it to our computation of similarity. Thus, instead of iterating through $M * N$ ratings, where M and N stand for the number of ratings of the first and the second user, we only iterate through $min(M, N)$ ratings and make $min(M, N)$ dictionary lookups.

Implementation Improvements We changed all of the usages of immutable data structures to mutable ones. Later on, we decided to use Arrays instead of Maps to store information about users, as it is stated that the dataset is compact. Therefore, we assumed that each user id represents an index in the Array. As such, we significantly improved the lookup time. Some useful data structures that we're using throughout the tasks are `ratingsByUser` and `ratingsByItem`, which are Arrays of Maps. For the `ratingsByUser`, the Array indices are the users, while the Map keys are the items and the values are the ratings. Initially, we populated these Arrays by using `groupBy` on the training dataset, by user or item. Currently, we initialize them by iterating through the dataset and adding them to the corresponding dictionary. We removed all *map* operations and replaced them with `for` loops and `while` loops. With these changes, we achieved a time of 2321.8027 ms.

We summarize all of the improvements we made in table 9

Listing 3: Computing and storing the denominator of all ratings for preprocessing

```
val norm2RatingsByUser = normalizedRatingsByUser.mapValues(x =>
    sqrt(x.map(y => pow(y.rating, 2)).sum))
```

Listing 4: Computing and storing the denominator of preprocessed ratings on-demand

```
if (norm2RatingsByUser(user) != -1) {
    // If the denominator was already computed, return it
    return norm2RatingsByUser(user)
}

// Compute the denominator
var norm2 = 0.0
for (rating <- getNormalizedRatingsByUser(user).values) {
    norm2 += pow(rating, 2)
}
norm2 = sqrt(norm2)

// Store the denominator for later use
norm2RatingsByUser(user) = norm2
norm2
```

Task	Value
Similarity User 1 - User 1	0.0000
Similarity User 1 - User 864	0.2423
Similarity User 1 - User 886	0.0000
Prediction User 1 Item 1	4.3190
MAE	0.8287

Table 7: Metrics for k-NN predictor with k = 10.

k	MAE
10	0.8287
30	0.7810
50	0.7618
100	0.7466
200	0.7400
300	0.7391
400	0.7391
800	0.7392
943	0.7372

Table 8: MAE based on various values of k in the nearest-neighbours implementation.

6 Recommendation

The recommendations are generated by looking at the movies that were not rated by a user, then computing the predicted ratings for those movies. Afterwards, we sort them by predicted rating and select the top movies.

6.1 Questions

R.1 The prediction of the k-NN predictor with additional ratings from user ”944”, using Adjusted Cosine Similarity and $k = 300$ can be seen in table 10.

R.2 Table 11 presents the top 3 movie recommendations for user ”944”.

References

[1] Milestone 1. `https://gitlab.epfl.ch/sacs/cs-449-sds-public/project/cs449-Template-M1/-/blob/master/Milestone-1.pdf`.

Improvement	Time (ms)	StdDev (ms)
No improvement	98496.4487	1755.7624
Faster intersection in Adjusted Cosine Similarity *Instead of chained functional calls, iterate through all pairs of user ratings and check whether they refer to the same item.*	37673.5687	3483.9449
Delayed computation & Storing results *Preprocessing denominator* *Average user rating* *Normalized ratings by user/item* *User similarity* *k-nearest neighbours*	13106.2385	1066.0837
Improved k-nearest users search	7470.7791	710.6451
Even faster intersection in Adjusted Cosine Similarity *Iterate ratings of one user and check if they have been rated by the other user through Map lookup.*	5513.0615	873.8009
Implementation improvements *Change immutable objects to mutable ones* *Change Maps to Arrays when feasible* *Replace functional calls (map/groupBy/filter) with for-loops and while-loops*	2321.8027	312.5826

Table 9: Changes and their impact on running time for k-NN.

Task	Value
Prediction User 1 Item 1	4.1321

Table 10: Prediction of the Recommender model.

Movie Id	Movie Name	Predicted Rating
119	Maya Lin: A Strong Clear Vision (1994)	5
814	Great Day in Harlem	5
1189	Prefontaine (1997)	5

Table 11: Top 3 recommendations for user 944.