Advances in Data Mining: Assignment I.

Haoran Ding
bitdhr@hotmail.com

Petra Kubernatova
pkubernatova@gmail.com

1 Introduction

Our task within this assignment was to implement several recommendation algorithms that have been discussed during the course, and estimate their accuracy with: the Root Mean Squared Error (RMSE) and the Mean Absolute Error (MAE) with help of 5-fold cross validation.

2 Input Data

The first step when implementing these algorithms is to load the data from the 1M data file downloaded from the MovieLens website into a numpy array with 1000209 rows and 3 columns. We do this using the numpy.genfromtxt() function. When this is done, we get a numpy array called *ratings* that looks something like this:

userid	movieid	Rating
1 *ratings[0,0]*	1193 *ratings[0,1]*	5 *ratings[0,2]*
...	...	...
6040 *ratings[1000209,0]*	1097 *ratings[1000209,1]*	4 *ratings[1000209,2]*

Table 1: Snapshot of the *ratings* array

According to the specifications in the readme.txt file of the downloaded dataset, the userid ranges in values from 1 to 6040 while the movieid ranges from 1 to 3952.

3 Naive Algorithms

The implementation of these algorithms which were introduced during the lecture was mostly very straightforward.

3.1 Average Rating

$$R_{global}(user, movie) = mean(all\ ratings)$$

Task: Calculate the average of all ratings in the data set

To calculate the global average rating, we simply take the third column of our *ratings* numpy array and use the numpy.mean() function to calculate it.

For this particular dataset, the result was: **3.58156445303**

3.2 Average User Rating

$$R_{user}(user, movie) = mean(all\ ratings\ for\ user)$$

Task: Calculate the average of all ratings given by a user

To calculate the average of all ratings given by a particular user, we go through each userid, find all rows in our *ratings* array that contain this userid and calculate their mean using the numpy.mean() function.

3.3 Average Movie Rating

$$R_{movie}(user, movie) = mean(all\ ratings\ for\ movie)$$

Task: Calculate the average of all ratings given to a movie

To calculate the average of all ratings of a particular movie, we go through each movieid, find all rows in our *ratings* array that contain this movieid and calculate their mean using the numpy.mean() function.

3.4 Linear Combination

$$R_{user-movie}(user, movie) = \alpha * R_{user}(user, movie) + \beta * R_{movie}(user, movie) + \gamma$$

Task: Predict movie ratings using linear regression with the average user rating and average movie rating.

To find the parameters α, β and γ, we used the numpy.linalg.lstsq() function. However, before being able to use this, we needed to create a new input numpy array (*ratings_new*). This array is very similar to the previously made *ratings*, but the userid column is replaced by $R_{user}(user, movie)$ and the movieid column is replaced by $R_{movie}(user, movie)$. You can see a representation of this array in Table 2.

$R_{user}(user, movie)$	$R_{movie}(user, movie)$	Rating
3.713 *ratings_new[0,0]*	3.476 *ratings_new[0,1]*	1 *ratings_new[0,2]*
...	...	...
3.581 *ratings_new[1000209,0]*	2.4 *ratings_new[1000209,1]*	1 *ratings_new[1000209,2]*

Table 2: Snapshot of the *ratings_new* array

3.4.1 Accuracies achieved

The approach that we used to test our implementation is the so-called 5-fold cross-validation scheme. This means that we split our available data at random into 5 parts of more or less equal sizes, developed 5 models for each combination of 4 out of 5 parts. Next, we applied each model to the part that was not used in the training process. From this we got 5 different estimates of the accuracy (RMSE, MAE) which we averaged to get a good estimate of the error on possible future data. We used 17 as the random seed. Table 3. shows the results that we achieved.

Result Table

Fold no.	α	β	γ	RMSE	MAE
1	0.781	0.874	-2.349	0.909	0.720
2	0.782	0.875	-2.356	0.910	0.720
3	0.782	0.875	-2.355	0.909	0.719
4	0.781	0.874	-2.349	0.908	0.718
5	0.781	0.874	-2.348	0.912	0.722
Average				**0.909**	**0.720**

Table 3: Results of the linear combination algorithm

Size of the test set was 200042 elements and size of the training sets was 800167.

3.4.2 Fall-back values

When building models for "average user rating" and "average movie rating", we had to take into account that during the sampling process some users or some movies will disappear from the training sets - all their ratings will enter in the test set. In our approach we replaced these non-available values with the average of all ratings in that particular set.

3.4.3 Time and memory consumption

Time consumption

We have counted that the time consumption of our implementation is $O(n^2)$. We calculated it in this way:

$$O(f_s(n)) + F * (U + D + O(f_{lr}(n)) + \frac{1}{5}N)$$

$O(f_s(n))$ = time complexity of the shuffle function, which is O(n)
$O(f_{lr}(n))$ = time complexity of the linear regression function, which is $O(n^2)$
N = total number of ratings, which is 1000209
F = number of folds, which is 5
U = total number of users, which is 6040
D = total number of movies, which is 3952

When we ran the implementation if took 4 minutes and 43 seconds to complete all 5 folds.

Memory consumption

We have counted that the memory consumption of our implementation is $O(n)$, where n is the number of records in the data set, which is 1000209.

3.4.4 Experiments

The assignment instructions state that we should try to improve predictions by rounding values bigger than 5 to 5 and smaller than 1 to 1 (valid ratings are always between 1 and 5). We have tried running our code both with the rounding and without and got the following results:

Average RMSE without rounding 0.910110558546

Average RMSE with rounding 0.909958269241

Average MAE without rounding 0.720884689706

Average MAE with rounding 0.720338084574

As you can see, the values for RMSE and MAE are better when rounding to the prediction if applied.

3.4.5 Conclusion

Overall, our implementation of the linear combination algorithm was quite successful in comparison to the RMSE and MAE values we have available from the MyMediaLite website. However, it runs a bit slow. It would be beneficial to go through the code again and see how it can be made more effective. For example reducing the amount of for loops, which slow down the script. Furthermore, we found out that rounding the prediction so that it is in the 1-5 interval improves both RMSE and MAE.

4 Matrix Factorization with Gradient Descent and Regularization

Task: Implement the Matrix factorization with Gradient Descent and Regularization algorithm to predict ratings given by a user to a movie

Matrix factorization can be used to discover latent features underlying the interactions between two different kinds of entities. In our case these two entities are users and movies. Given that each users have rated some items in the system, we would like to predict how the users would rate the items that they have not yet rated, so that we could make recommendations to the users. All the information we have about the existing ratings is represented by a matrix containing userids in its rows and movieids in its columns. The elements are the ratings given to the particular movieid by a particular userid. However, some elements from this matrix are missing, so we need to fill those blanks with our predictions. The result of the matrix factorization will be a matrix with no missing elements.

We started the implementation of this algorithm by creating an input matrix. This matrix had the size no. of users x no. of movies. This came out to be 6040 rows and 3952 columns. We created a two-dimensional numpy array (mf) which had userids in its rows and movieids in its columns. In Figure 1. you can see a snapshot of the "matrix".

$$mf = \begin{pmatrix} rating_{userid=1,movieid=1} & \cdots & rating_{userid=1,movieid=3952} \\ \vdots & \ddots & \vdots \\ rating_{userid=6040,movieid=1} & \cdots & rating_{userid=6040,movieid=3952} \end{pmatrix}$$

Figure 1: Snapshot of the input matrix for the matrix factorization

The next steps of the implementation were pretty straightforward. For the sake of this report not being too long, we are not going to explain the algorithm itself. We used the paper by Takacs et al. as our source, so all information can be found there.

4.1 Accuracies achieved

In our implementation we used the same cross-validation scheme as for the linear combination algorithm. See Section 3.4.2 for details.

We achieved the following results while running our implementation with the following parameters:

Number of factors = 10
Number of iterations = 75
Regularization = 0.05
Learning rate = 0.005

Number of factors is the number of latent features we are looking for. Number of iterations is how many times we perform the learning process. Regularization is used to avoid overfitting. The learning rate is a constant whose value determines the rate of approaching the minimum.

Result Table

Fold no.	RMSE	MAE
1	0.879	0.688
2	0.879	0.688
3	0.879	0.689
4	0.877	0.686
5	0.878	0.687
Average	**0.879**	**0.688**

Table 4: Results of the matrix factorization algorithm

4.2 Time and memory consumption

Time consumption

We have counted that the time consumption of our implementation is:

$$O(FSUDK)$$

F = number of folds, which is 5
S = number of steps, which is 75
U = total number of users, which is 6040
D = total number of movies, which is 3952
K = number of factors, which is 10

Memory consumption

We have calculated that the memory consumption of our implementation is $O(UD)$, where U is the number of users, which is 6040 and D is the number of movies, which is 3952.

4.3 Conclusion

We experienced big issues with this algorithm. The implementation itself was very straightforward, especially using the Takacs et al. paper. However, the results that we achieved were very disappointing. The average RMSE and MAE values that we got were quite satisfactory compared to the results stated on the MyMediaLite website. They achieved an RMSE of 0.857 and MAE of 0.675 compared to our RMSE of 0.879 and MAE of 0.688. This being said though, our implementation lacked severely in the speed department. One fold took over 1.5 hours to complete. The whole run took approximately 6.5 hours to complete. These results were confirmed by our time consumption calculation. This prevented us from carrying out experiments with different parameters, because we simply did not have enough time to complete them. We have searched for the cause of the slowness of our implementation and believe that the main cause is the use of a lot of for loops. For loops (and especially nested ones, which we have a lot of) slow down the script by a significant amount. For this reason, we would opt for operations with matrices in possible future implementations instead of using for loops. We strongly believe that this would speed up the run of the program significantly.