Assignment 3
Big Data Analytics Programming

Evangelos Ntavelis
r0692337
evangelos.ntavelis@student.kuleuven.be

April 15, 2018

Contents

1 Introduction

In this assignment we will discuss the performance improvement we can achieve by using indices on database queries and the implementation and performance issues that arise when building a Neighbor-Based Collaborating Filtering System for Movie Recommendations.

2 Index Structures

In order to minimize the effect of randomness on the performance of the naive and index based approaches we run the queries one million times, the average time for each query is shown in the Figure 1. We also checked that indeed the two queries produce the same results. Let's discuss the results:

- **Queries 1 & 2:** we observe that the index solution takes 0.9% and 1.6% of the naive solution time respectively. The huge improvement makes sense, as when constructed the BitMap we already made the selection, as bits in the attribute columns. Thus selections and their counts are trivial.

- **Query 3:** the use of the BitSlice Index gives us 0.3% of the naive running time. Comparing with the naive solution here is kind of cheating as the numerical values are stored as Strings we have to parse, a process that is already done for the index. At the same time, the BitSlice structure permits us to easily sum values based on the number of bits we have on each bit column.
- **Query 4:** the fraction here is 4.3%. We also observe the cumbersome double parsing, yet applying the filter increased the BitSlice index time.
- **Queries 5 & 6:** The naive's double parsing here takes less time as it is applied only to a fraction of the records, as the selection happens first. We observe the indices approach to take 22.4% and 17.4% of naive's time accordingly.

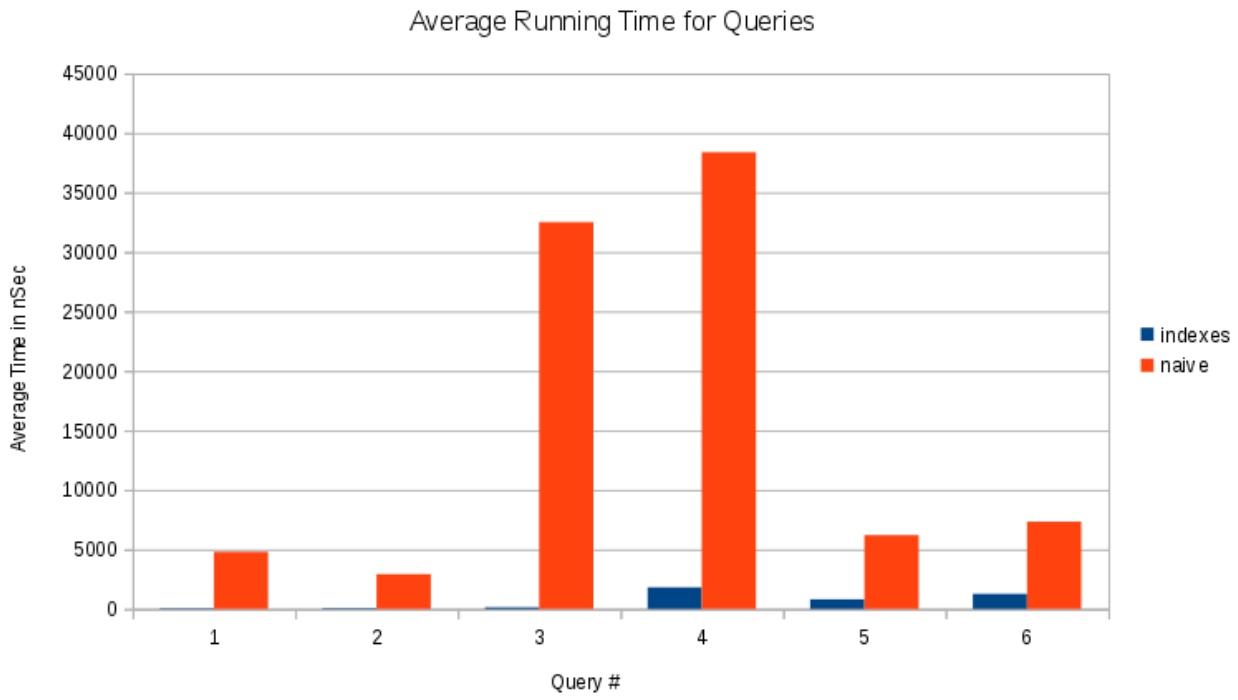

Figure 1: Query Performance

3 Movie Recommendations

In this part we will describe the decisions we made when implementing the simple Collaborating Filtering model for recommending movies.

3.1 Fitting the dataset into 2 Gigabytes of RAM

The first issue we encounter when implementing the model was how to fit the Pearson Correlation Matrix in the limited memory space. Progressively, we found ways to lessen the matrix memory usage:

- **Symmetry:** The Pearson Correlation Formula is symmetrical and so is the corresponding matrix, we could omit storing the diagonal, as we always know that the correlation of an element to itself is 1, and all the values below it.

- **Values Range & Precision:** The Pearson correlation takes values between -1 and 1; our aim is to store the number rounded to four decimal points. In total we have 20001 numbers, which could fit into a *short* variable type and thus use only 2 bytes per stored correlation.

Up to this point we were able to significantly reduce the needed space, but we were not nearly there. Next step, choice of **Data Structures**. Firstly we developed a wrapper class for user-correlations, afterwards we discovered that the *Neighbors* was a essentially the same idea. Feeling both good and bad with ourselves we adjusted our(?) creation. We also noticed that the most of the Correlation Matrix Values are *NaNs*.

- **Arrays** Redundantly stores the NaNs and consumes about 5G of RAM

- **Hashmap** At first we used a Hashmap of ¡Integer, Neighbor¿which later upgraded to an Array of Hashmaps ¡Integer, Correlation¿. Both of these implementations consumed more memory than the limit.

- **Arraylists** We ended up using an Array of Arraylists. The problem with this implementation is that the retrieval is not of constant time, which affected the way we implemented the prediction later.

Proud with our ideas, we set to compile and run. The sun out of our window was warmly smiling at us, the sky was the bluest we've encountered the last six months below the eternal gray sky of Belgium. The *outside window* was telling me to go out, the *screen window* was telling me *Memory Error*. So, to further limit our memory usage we made the following choices:

- **Least Common Rated Movies:** the Pearson Correlation formula for 2 items produce only -1 or 1, for 3 we are unable to compute the significance intervals. Generally the less this threshold is, we have more noise in our correlation matrix. We experimented with different values and had the correlation of two user pairs without enough items set to *NaN*.

- **Correlation Threshold:** In Neighbor-Based Collaborating Filtering methods there are two main approaches: set a maximum number of Neighbors or set a minimum threshold to correlation values used for predictions. By setting this limit at the Matrix Computation Stage, we were able to further reduce the usage.

At last we made it!

3.2 Code Performance Choices

Generally, we tried to omit doing any extra calculations.We made the following note-worthy choices:

- The first way to achieve this was to pre-compute the average user ratings, store them and pass them when we need them.
- We had the userToRatings arraylists sorted so we can retrieve the same rated movies in linear time, instead of the square time of java's containsAll. We also first made the check for same items and then the computations to save time.
- When writing the csv file, we used indices stored in an array in order to not transverse our Correlation Lists only two times.
- We batched the prediction on an per user basis, and refilled the missing half of our matrix on the go without any additional memory needs (When visiting the correlation pair i,j, we stored the pair to j,i and after making the predictions for user i we cleared his correlation list.)

3.3 Parameter Tuning and Evaluation

Having the Pearson Correlation Matrix at hand, we use the following formula to make our predictions:

$$P_{ik} = \bar{R}_i + \frac{\sum W_{ij}(R_{jk} - \bar{R}_j)}{\sum W_{ij}}$$

Firstly we tried to change the baseline estimation to incorporate the global average and the rating deviation of users and movies but we found that the user average produces better predictions.

The parameters we had to fine tune were the following: additionally to least common rated movies and the correlation threshold we set the Adjustment Threshold to lower the significance of correlations based on few co-rated items. After running several experiments on parallel we produced the results shown on Figure 2. We found the our lowest value for RMSE is 0.9538 for threshold set to 0.4, least common items to 5 and a adjustment threshold at 15.

We tried using a negative threshold as well by taking the negative correlation values lower than it into account. We thought that extremely dissimilar users choices would lower the error, but that was not the case. We also tried to use a Nearest Neighbors approach, but as shown on Figure 3 the results were not improved and we the added sorting time was not justified.

The memory limitation is the biggest constraint to our performance, the blank cells in our tables are the ones that we could not fit the Correlation Matrix in 2 Gigabytes of RAM. Closer we are to their configuration, the better the results we produced.

We have to note that the Pearson Correlation Matrix was produced with the Threshold Collaborating Filtering model in mind and not the kNN one. Thus, by only saving the most relevant users at the Matrix Computation phase we may could produce better results.

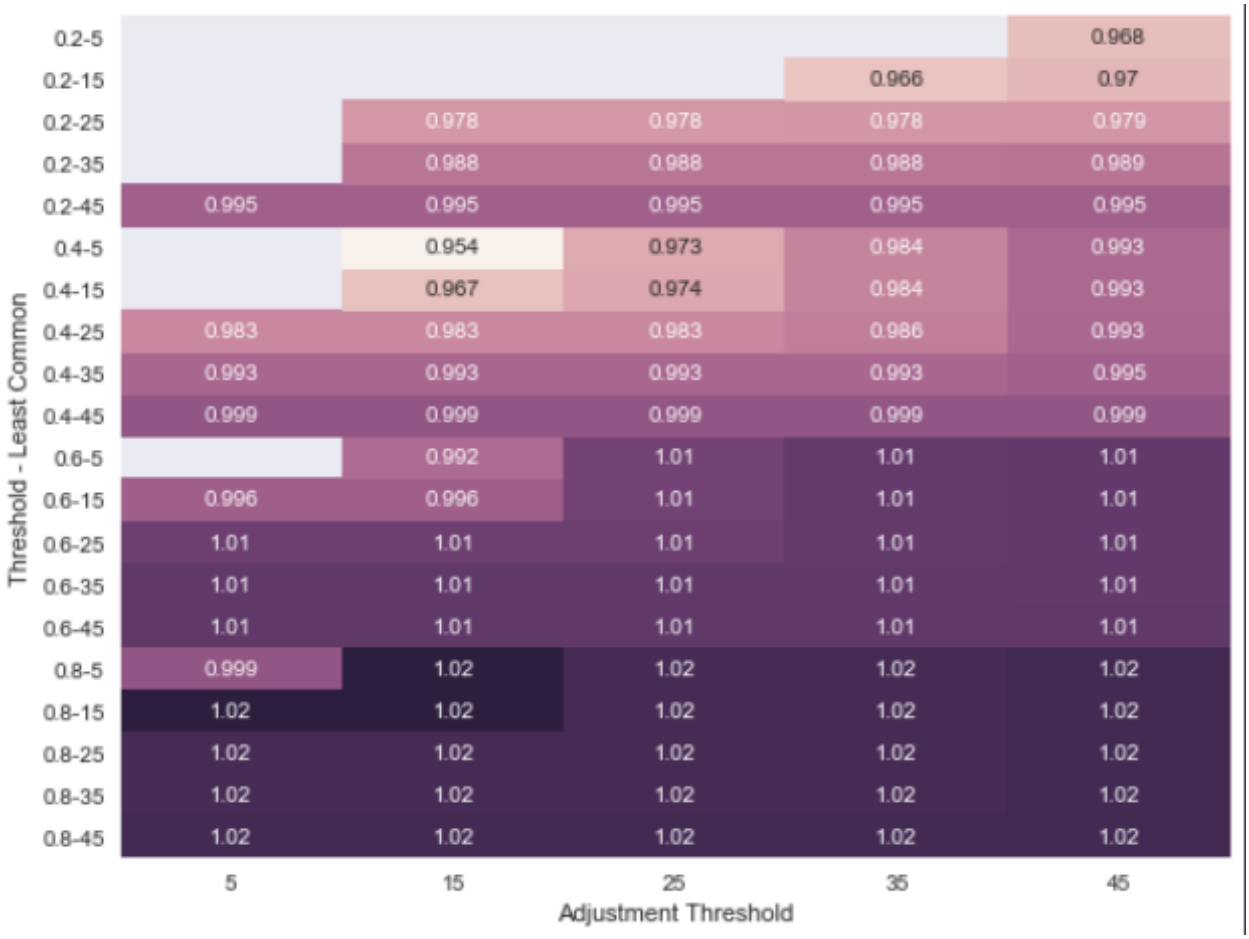

Figure 2: RMSE: experimental results

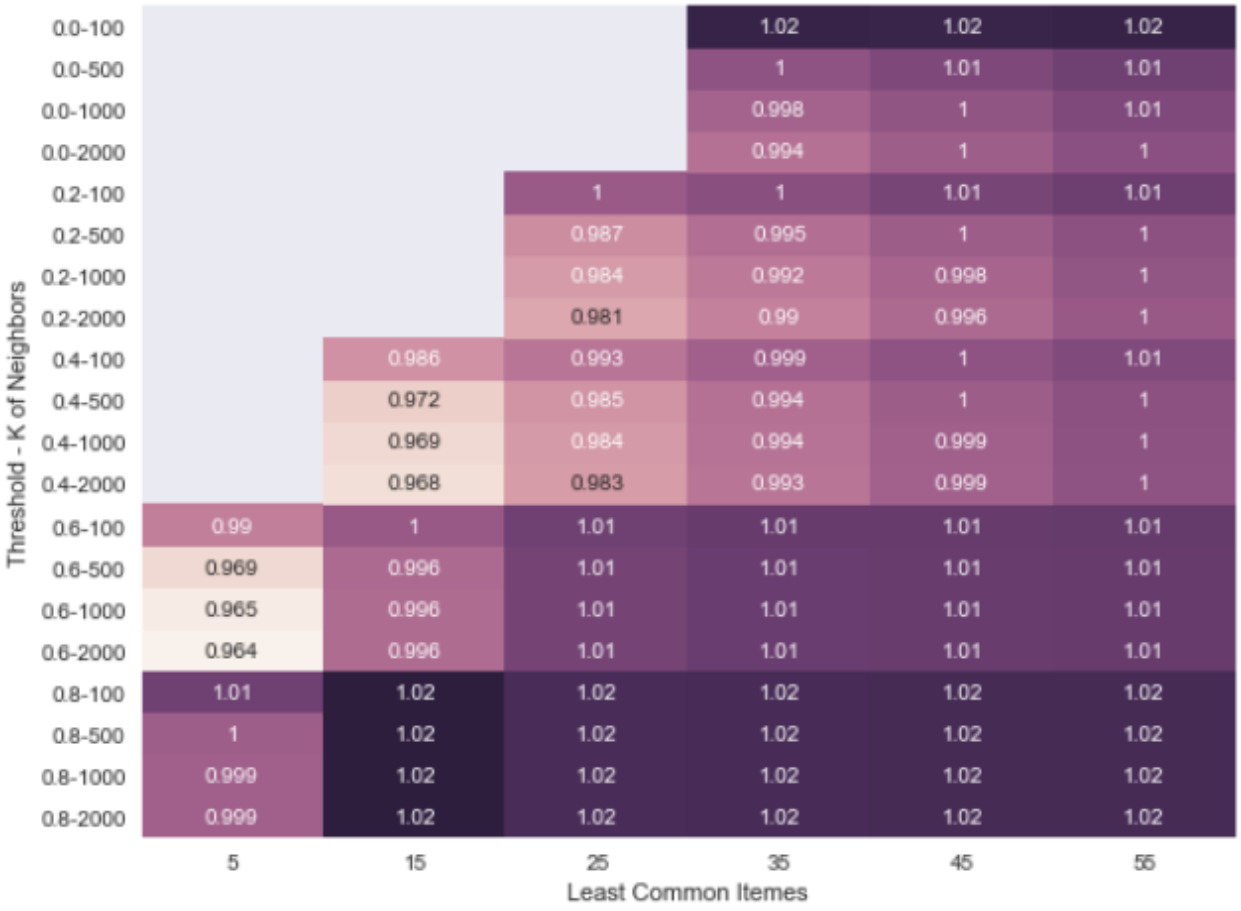

Figure 3: Nearest Neighbors RMSE: experimental results