CS449 - Project Milestone 3 (Report)

Saoud Akram - SCIPER : 273661

Remark : In this report, the term "movie" and "item" are used interchangeably.

Part 3.2

1. Reimplement the Predictor of Milestone 2 using the Breeze library and without using Spark. Test your implementation on the 'ml-100k/u1.base' as a training set, and the 'ml-100k/u1.test' as a test dataset, and k = 200

- MAE for $k = 100$: 0.75635
- MAE for $k = 200$: 0.74845

With regards to the implementation, we computed the similarities by first pre-processing the ratings (i.e. we had a matrix of pre-processed ratings) which we multiplied by itself (transposed) and then filled the diagonal with zeroes in order to nullify the self-similarities. Then, to filter the top k neighbours, we iterated over each rows and kept the k highest similarities. After that, in order to find the MAE, we first computed the user-specific weighted-sum deviation for an item i for every pair (u, i) in the test set and then every predictions.

2. Measure the time for computing all k-nearest neighbours (even if not all are used to make predictions) for all users on the ml 100k/u1.base' dataset. Output the min, max, average, and standard deviation over 5 runs in your implementation.

min [s]	max [s]	average [s]	stddev [s]
2.2782	2.5484	2.3950	1.0699

(Please note that the timings were converted into seconds and rounded to the 4th decimal, as it allows for better readability. For more precise answers, please consult the appropriate .json file which contains the exact timings in μs)

Note that here we took $k = 200$ to get those answers

3 . Measure the time for computing all 20,000 predictions of the 'ml-100k/u1.test'.Output the min, max, average, and standard-deviation over 5 runs in your implementation.

min [s]	max [s]	average [s]	stddev [s]
5.6739	7.3257	6.4165	0.5557

(Please note that the timings were converted into seconds and rounded to the 4th decimal, as it allows for better readability. For more precise answers, please consult the appropriate .json file which contains the exact timings in μs)

Note that here we took $k = 200$ to get those answers

4. Compare the time for computing all k-nearest neighbours of 'ml-100k/u1.base' to the one you obtained in Milestone 2 (Q.2.2.7). What is the speedup of your new implementation (as a ratio $\frac{t_{old}}{t_{new}}$)? Why do you think that is the case?

Considering Erick Lavoie's answer on Moodle to our question (https://moodle.epfl.ch/mod/forum/discuss.php?d=61789), we re-ran our code of Milestone 2 to compute the the top $k = 200$ similarities per (and record the timing taken by this process). In other words, I computed **all** similarities (i.e. computing n^2 similarities where $n = |U| = 943$) and then filtered the 200 best per user. As such, the average time I recorded for computing the top $k = 200$ similarities is 31.85 seconds using the M2 implementation. Therefore, the speedup we obtain is given by $\frac{31.85}{2.4} \approx 13.27$, which is higher than 10.

Why do we observe such an improvement ? We're harnessing the power of linear algebra instead of relying on long "join" operations, which as we pointed out in M2, is what was most costly time-wise. Also, using a library like Breeze (and the data structures it provides such as Sparse Matrix and so on) optimizes the way we're using the memory and how we're doing the computations (typically, the matrix multiplication operation that the library offers is much faster than a naïve double loop implementation for matrix multiplication, which is definitely helpful).

Remark: As we pointed out, we did not take the answer we got in Milestone 2 Q. 2.2.7. Why ? Because in question 2.2.7 of Milestone 2, we are asked to record the timings to compute **all** similarities (as if $k = 943$). According to Erick's answer to my question, to "compute all k-nearest neighbours of 'ml-100k/u1.base'" we have to take $k = 200$ (or, more precisely, take the k passed as an argument in the command line). Therefore, "comparing the time for computing all k-nearest neighbours of 'ml-100k/u1.base' to the one you obtained in Milestone 2 (Q.2.2.7) doesn't really make sense since we would be comparing the timings for two different tasks (namely, the first task computes the top 200 similarities per user and the second computes all similarities per user). That's why we re-ran our code in M2 to compute the top 200 similarities per user.

Also, consider the following. For $k = 943$, the optimized code in Milestone 3 part 3.2 takes on average 1.3 seconds to compute the k-nearest neighbours (i.e. the top $k = 943$ similarities) (You may check this by running my code and setting $k = 943$ in the command line). It takes less time than any other k (Why ? To compute the top k similarities per user, we first compute all similarities (i.e. as if $k = 943$) and then filter the top k similarities per user. In the case of $k = 943$, we can skip the filtering part since $k = 943 = \text{conf.users}() = |U|$ and bypassing the filtering part is a considerable advantage time-wise that only $k = 943$ has).

Now that we have the average time taken by our optimized code in M3 for computing **all** similarities (i.e. when $k = 943$, which means that the matrix has exactly $|U|$x$|U|$ active entries), we can compare it to the answer in Milestone 2 Q.2.2.7. Notice that in Milestone 2 Q.2.2.7, in order to save time, I stated that $s_{u,v} = s_{v,u}$ and therefore I didn't compute both of them but only one. I also didn't need to compute self-similarities. As such, I ended up computing $n(n-1)/2$ similarities where $n = |U| = 943$, which took me, on average, 9.3 seconds. Therefore, one could argue to compute all n^2 similarities, it would take $2 \cdot \frac{n^2}{n^2-n} \cdot 9.3 \approx 18.6$ seconds. Finally, we can compute the speedup which is given by $\frac{18.6}{1.3} \approx 14.3$, which is higher than 10

Part 4.1

1.Test your spark implementation with the ml-1m/ra.train as a training set, and the ml-1m/ra.test as a test dataset, and k = 200

With the ml-1m/ra.train as a training set, and the ml-1m/ra.test as a test dataset, the MAE we obtain for $k = 200$ is 0.734624743681846

Note that with ml-100k/u1.base and ml-100k/u1.test we obtain the same result we obtained in 3.2.1 for $k = 200$

2. Manually run your implementation on the cluster 5 times and compute the average. Compare the average kNN and prediction time to your optimized (non-Spark) version of Section 3 on the ml-1m/ra.train dataset.

Note that in the following exercise, we chose $k = 200$

For the Spark implementation, we got the following results

Attempt #	Duration for computing KNN [sec]	Duration for computing predictions [sec]
1	15.8607	70.9248
2	15.6391	72.8578
3	18.3742	72.875
4	18.6311	77.1767
5	17.221	73.9995

As such, the Spark implementation takes on average 17.15 seconds to compute the k-nearest-neighbours and takes approximately 73.57 seconds to compute the predictions

For the non-Spark implementation, we got the following results:

Attempt #	Duration for computing KNN	Duration for computing predictions
1	13.1659	72.546
2	11.0516	68.4863
3	12.9844	68.5025
4	11.2082	66.6693
5	12.1695	69.5590

Therefore, the average time taken by the non-Spark implementation to do compute the kNN is 12.11 seconds and the average time to compute predictions is 69.15 seconds. Note that I had to increase the heap size to be able to run the code on this dataset.

We can observe that the non-Spark implementation is faster. Why ? The non-spark implementation relies exclusively on linear algebra. We don't explicitly do any parallelization. That is, we don't do fork/joins of threads. On the other hand, with the spark implementation, we use `sc.parallelize(seq)` to create an RDD which we map to a certain function. This function, internally, does the fork/joins in order to parallelize the operation, which comes at the cost of an overhead (creating the threads, doing the forks/joins, broadcasting, etc.). On top of that, since we set the number of executors to 1, we don't gain anything from this overhead, as the entire work is done by a single executor. Also, once we're done doing the parallel work, we have rebuild a matrix from the results of all the parallel works, which also takes time.

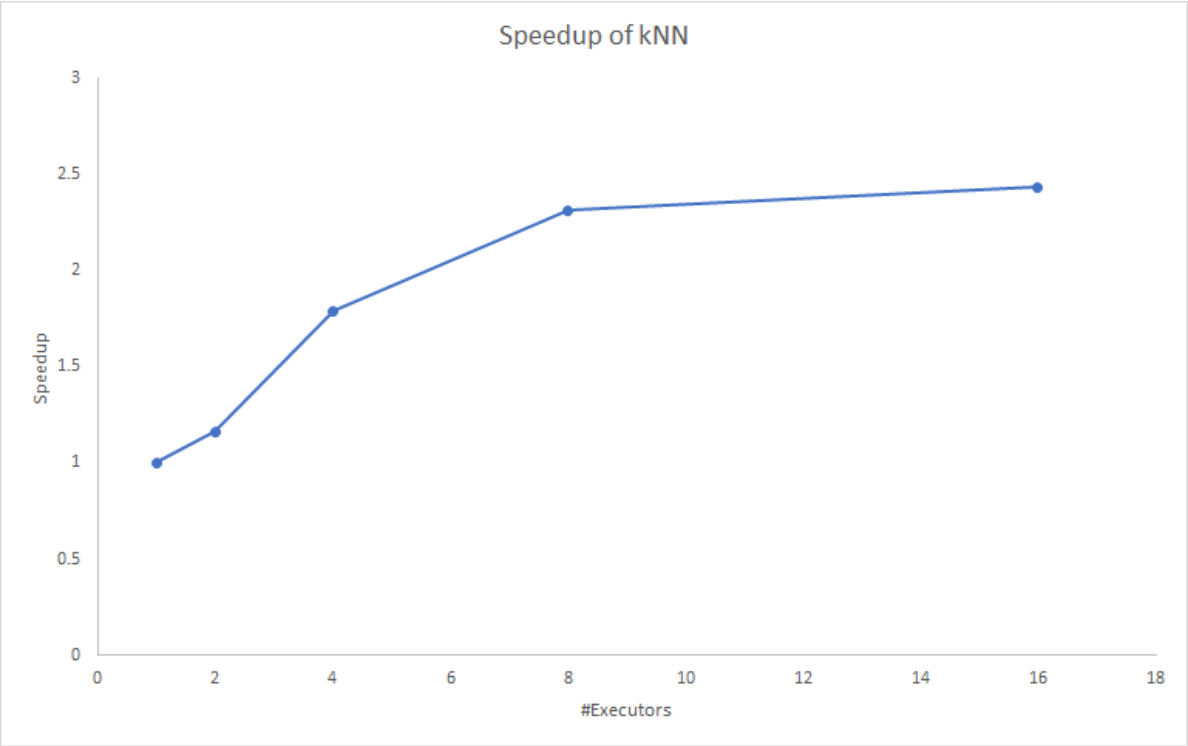
3. Measure and report kNN and prediction time when using 1, 2, 4, 8, 16 executors on the IC Cluster in a table. Perform each experiment 3 times and report the average, min, and max for both kNN and predictions. Do you observe a speedup? Does this speedup grow linearly with the number of executors, i.e. is the running time X times faster when using X executors compared to using a single executor?

Note that in the following exercise, we chose $k = 200$

The timings we got for KNN are the following :

#executors	1	2	4	8	16
min [s]	15.639	12.846	7.99	6.664	6.341
max [s]	18.631	14.216	9.134	6.859	6.527
avg [s]	15.639	13.478	8.742	6.767	6.434

The following figure depicts the speedup for computing KNN w.r.t. the number of executors



We see that the speedup isn't very linear, but has more of a logarithmic shape. It would seem that the maximum achievable speedup is around 2, which means that we parallelized approximately 50% of the entire computation that is done within the function (i.e. when computing the kNN).

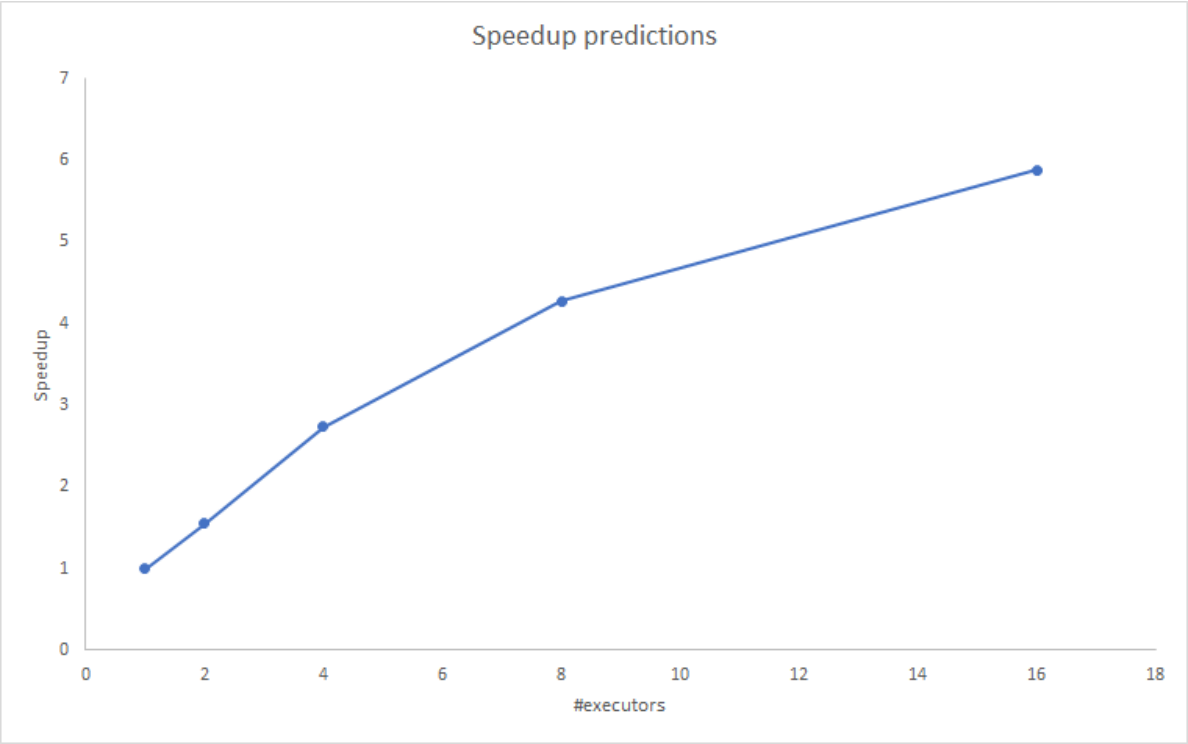
As such, if we assume that there is a price (say 50 CHF) for each executor, then purchasing 16 executors would amount 800 CHF for a speedup of around 2.4. However, with 8 executors (and half the price since it would amount to 400 CHF), we would get a speedup of 2.3. Our point is that it is not always necessary to try and maximize the number of executors, but rather have the right amount.

To increase the speedup, we would need to make a higher portion of the program parallelizable.

- The timings for computing predictions are the following :

#executors	**1**	**2**	**4**	**8**	**16**
min [s]	70.925	45.05	23.996	16.543	12.17
max [s]	72.875	47.639	27.761	17.513	12.368
avg [s]	72.219	46.426	26.42	16.889	12.269

The following figure depicts the speedup w.r.t the number of executors



Here, the curve looks a little more linear, although we can see that it starts to flatten out at 8 executors (i.e. the speedup gain from #executors = 8 is not as consequential as it was before, but still remains fairly descent). By the looks of it, it seems like the portion of our program that's been parallelized amounts to at least 80%. Again, in order to increase the speedup, we need to try and parallelize a higher portion of the program. A higher speedup leads to lower execution times, which is what we strive for (and also minimize the number of workers, since they usually come at a certain price). Note that it is not always possible to increase the portion of the program that can be parallelized, hence the importance of choosing the right number of executors.

Part 5

Here are the results for all the questions. With regards to how they were obtained, please consult the code :

Question	Result
Q5.1.1__MinDaysOfRentingICC.M7	1715.68627
Q5.1.1__MinYearsOfRentingICC.M7	4.70051
Q5.1.2__ContainerCheaperThanICC.M7	FALSE
Q5.1.2__DailyCostICContainer_Eq_ICC.M7_RAM_Throughput	20.896
Q5.1.2__RatioICC.M7_over_Container	8.27922
Q5.1.3__ContainerCheaperThan4RPi	FALSE
Q5.1.3__DailyCostICContainer_Eq_4RPi4_Throughput	0.184
Q5.1.3__Ratio4RPi_over_Container_MaxPower	0.29348
Q5.1.3__Ratio4RPi_over_Container_MinPower	0.0587
Q5.1.4__MinDaysRentingContainerToPay4RPis_MaxPower	16.5613
Q5.1.4__MinDaysRentingContainerToPay4RPis_MinPower	16.43729
Q5.1.5__NbRPisForSamePriceAsICC.M7	369
Q5.1.5__RatioTotalRAMRPis_over_RAMICC.M7	1.92188
Q5.1.5__RatioTotalThroughputRPis_over_ThroughputICC.M7	3.29464
Q5.1.6__NbUserPerGB	138888.8889
Q5.1.6__NbUserPerICC.M7	106666666.7
Q5.1.6__NbUserPerRPi	555555.5556

As for question 7, which concerns which is my preferred option, I would go for the 4Rpi. I proceeded by elimination. For the same amount of resources, the container seems to be the most expensive. Since we would ideally like to have a long term solution to this, containers are not the best option. Whilst the ICC.M7 may seem like a good choice, it seems to be offering much more than we require (note that we keep k=200 and a user has on average 100 ratings), which means that we don't require such a large infrastructure, and therefore don't need to pay a high price. So we chose 4Rpi. We would chose to buy it, because it isn't that expensive and fits our needs best. After buying, we only need to pay a small fee, which is diluted over time. Whilst it may be true that it may require more maintenance, it still the best cheapest and the one that is on par with our requirements (not to mention, it also better for the environment).