ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

# PROJECT MILESTONE 2: OPTIMIZING, SCALING, AND ECONOMICS

CS-449 SYSTEMS FOR DATA SCIENCE

Irina-Madalina Bejan
Siran Li

20th May 2022

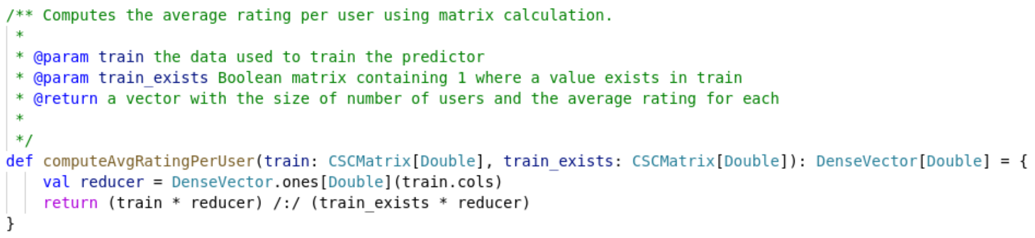
# 3 Optimizing with Breeze, a Linear Algebra Library

## BR.1 Predictor using the Breeze library

Using the Breeze library to reimplement the kNN predictor, we tried to optimize the running time, while preserving the correctness of the algorithm. The output values were the same as in Milestone 1:
(1) similarities between user 1 and itself: 0;
(2) similarities between user 1 and user 864: 0.24232304952129619;
(3) similarities between user 1 and user 886: 0;
(4) prediction for user 1 and item 1: 4.319093503763853;
(5) prediction for user 327 and item 2: 2.6994178006921192;
(6) mean absolute error (MAE): 0.8287277961963542

In our optimizations, we followed the guidelines given in the report and noticed slight improvements when applying the argtopk method on a DenseVector, when favoring foreach calculations and map operations over for iterations. Most importantly, the possibility of iterating through the active values of the sparse matrix instead of iterating through a dense, but rather empty space, brought the most improvement.

A slight optimization we considered is using 1-column vector filled with 1 to compute summations efficiently in a vectorized form rather than iterating through the active values, as illustrated below.



```
/** Computes the average rating per user using matrix calculation.
 *
 * @param train the data used to train the predictor
 * @param train_exists Boolean matrix containing 1 where a value exists in train
 * @return a vector with the size of number of users and the average rating for each
 *
 */
def computeAvgRatingPerUser(train: CSCMatrix[Double], train_exists: CSCMatrix[Double]): DenseVector[Double] = {
    val reducer = DenseVector.ones[Double](train.cols)
    return (train * reducer) /:/ (train_exists * reducer)
}
```

**FIGURE 1:** Optimizations using vectorized reducer.

Computed MAE of kNN predictions with different k values, the results are shown in the following table (rounded to 4th decimal place):

| | **k=10** | **k=300** |
|---|---|---|
| **MAE** | 0.8287 | 0.7392 |
| **Running time (s)** | 0.721 | 4.1 |

**TABLE 1:** Mean absolute error of kNN predictions and running times with different k values.

From the comparison, we can know that with a higher k value, the kNN predictor can keep more effective information for predicting more accurately. Therefore, the MAE with k=300 is lower than the MAE with k=10.

## BR.2 Computing time for all k-nearest neighbours

After optimizing with Breeze library, the computation time reported in the following table is the average time over 3 runs on '100k' datasets. We can compare the time with Milestone 1 we used to show the speedup and we ran both on the same personal machine for a fair comparison.

**Discussion of results:** From the above time results, we can get that the speedup is actually not significant for computing MAE between two milestones, as in our run the time seems to be around the same ballpark values.

| | k=300 | |
|---|---|---|
| | average(s) | stddev(s) |
| **Milestone 1** | 3.7s | 0.129s |
| **Milestone 2** | 4.1s | 0.207s |

**TABLE 2:** Comparison of computation time between Milestone 1 and Milestone 2.

We remember however, our initial implementation of KNN in Milestone 1 was close to 80s, thus if we didn't attempt to optimize it, we could have seen a great improvement. Our initial implementation of Milestone 1's KNN was slower due to several drawbacks: as we were using just Scala and not Spark, we didn't manage to get full benefits of the possible parallelization strategies and this was also making it hard to scale our solution for a higher amount of users and items, as some operations were still done iteratively.

However, as we worked to optimize the previous Milestone's KNN, we actually rewrote the operations in a vectorized form, not far from the current milestones and implemented by hand all operations, so it makes sense why the speedup is not significant, since we basically just switched our solution to using a library for matrix operations. A natural development from our previous work was using a DenseMatrix for the Milestone 2 which we noticed to bring higher speedup (2x), as it resulted in a time of 1.5-2s, and we hypothesize it is because the library is highly optimized compared to our attempt.

Because we were restricted to use the CSCMatrix to better support scaling up the solution for sparse inputs, we managed to get a an effective solution that runs in similar time based on efficient vectorized implementations, but we have now the freedom to scale up and run our code on much bigger and sparse datasets.

## 4 Parallel k-NN Computations with Replicated Ratings

### EK.1 Testing spark implementation

As we implemented the distributed version of kNN predictor, we checked using 2 workers the correctness of the results, and the output values were the same as in Section 3:
(1) similarities between user 1 and itself: 0;
(2) similarities between user 1 and user 864: 0.24232304952129619;
(3) similarities between user 1 and user 886: 0;
(4) prediction for user 1 and item 1: 4.319093503763853;
(5) prediction for user 327 and item 2: 2.6994178006921192;
(6) mean absolute error (MAE): 0.8287277961963542

To improve the time used by the exact and approximate variants, we followed the pseudocode presented and further parallelized the computation of the MAE for each user, item, by materialized the activeIterator over the test matrix, which saved a few seconds of the computations.

We also noticed a few edge cases when retrieving the topk values for one partition, first being when the value K is higher than the number of values in partition, which could lead to error. Secondly, we noticed negative similarities which can also be chosen as part of top-k when the number of positive values is lower than K. Thus, we decided to keep positive only values, meaning than in specific configurations the similarity for one user might depend on less than K neighbours, but we believe it resembles better the expected behaviour of the algorithm in practice.

### EK.2 Comparison of execution time

Running the distributed version on a single executor 3 times, we can compare the results with the optimized implementation () on the same dataset (rb.test) with k = 300 in table 3.

| | average(s) | stddev(s) |
|---|---|---|
| **Optimized** | 387.19 | 4.73 |
| **Distributed 4 workers** | 119.75 | 1.32 |

**TABLE 3:** Comparison of computation time between optimized version and distributed version using 4 workers.

As expected, the distributed version highly speeds up the running time of the KNN algorithm, with a factor of almost 4, having a similar time, even a bit higher than the distributed one using a single executor.

After comparing the computation time of optimized distributed version, we measured the computation time of distributed kNN predictor when using 1, 2, 4 and a k=300.

| Executor | average(s) | stddev(s) |
|---|---|---|
| **1** | 353.73 | 4.44 |
| **2** | 210.30 | 6.29 |
| **4** | 119.75 | 1.32 |

**TABLE 4:** Comparison of computation time with distributed version, varying number of executors.

In addition to table displays, we also visualized the execution time by a line chart:

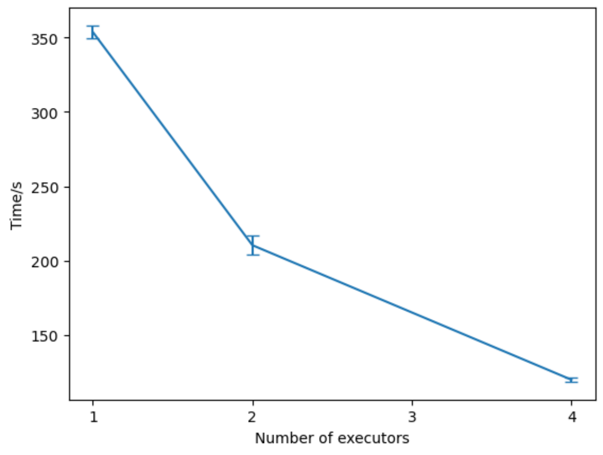


**FIGURE 2:** Figure of execution time with varying number of executors.

From the above table and figure, the speedup in execution time with increasing number of executors can be clearly observed. In the figure, this speedup grows linearly with the number of executors, and the running time X times faster when using X executors compared to using a single executor.

## 5 Distributed Approximate k-NN

### AK.1 Testing approximate k-NN implementation

Implemented the approximate kNN using the previous breeze implementation and Spark's RDDs with 10 partitions and 2 replications, k=10, the required output similarities are:
(1) similarities between user 1 and itself: 0;
(2) similarities between user 1 and user 864: 0;
(3) similarities between user 1 and user 344: 0.23659364388510976;
(4) similarities between user 1 and user 16: 0;
(5) similarities between user 1 and user 334: 0.19282239907090362;
(6) similarities between user 1 and user 2: 0;

A situation which required special care is when two users end up in different partitions and their similarity is retrieved by the algorithm multiple times, for which we choose the maximum value among the existing ones. This also gives us more flexibility and helped us optimize using RDD the combination of topk similarities retrieved for each partition, which greatly improves the time as the number of partitions and replicas goes up.

### AK.2 Comparison with varying replications

To vary the number of partitions in which a given user appears, we measured the MAE value and the number of similarity computations from distributed approximate kNN predictor with partition=10, k=300 on '100k' datasets, when setting replication to (1, 2, 3, 4, 6, 8) respectively. The results are showed in the following table:

| Replication | MAE | Partitions |
|---|---|---|
| **1** | 0.80876 | 10 |
| **2** | 0.75878 | 10 |
| **3** | 0.74588 | 10 |
| **4** | 0.74104 | 10 |
| **6** | 0.73920 | 10 |
| **8** | 0.73920 | 10 |

**TABLE 5:** Comparison of MAE values and number of similarity computation with different number of executors.

From Section, we also know that the MAE is 0.7392 (rounded to 4th decimal place) when using k=300 which actually matches the results we get with a high level of replication (6, 8).

We also notice that the minimum level of replication such that the MAE is still lower than the baseline predictor having a MAE of **0.7604** for k = 300 is 2, as the MAE with a replication of 2 is **0.75878**.

**Similarity computations:** The number of similarity computations required by the exact KNN depends on the number of users N and should be $N * (N-1)/2$ given the similarity is symmetric. When we run the approximate version of the KNN, we require a number of $(N/partitions) * (N/partitions - 1)/2 * replications$ computations. Thus, the ratio the approximate version reduces the number of similarity computations is $partititions^2/replications$. In our specific case, we can get the same result with 50 times less similarity comparison compared to the baseline KNN (Milestone 1) and 12.5 times less similarity computations compared to the exact KNN. However, it is possible different configurations of partitioning can reduce the number even further.

### AK.3 Comparison of execution time

We measured the MAE value, average and standard-deviation of execution time over 3 times from distributed approximate kNN predictor with partition=8, replication factor=1, k=300 on '1m' datasets, when using 1, 2, 4 workers respectively. The results are showed in the following table:

| Executor | MAE | average time(s) | stddev time(s) |
|---|---|---|---|
| **1** | 0.7446 | 137.39 | 2.39 |
| **2** | 0.7446 | 96.87 | 3.9 |
| **4** | 0.7446 | 55.82 | 1.1 |

**TABLE 6:** Comparison of MAE values, average and standard-deviation of execution time with different number of executors.

We notice that the approximate computations using replicated ratings brings 2-3x speed improvements over the exact k-NN with a comparable number of workers, while the MAE is also lower.

## 6 Economics

### 0.1 E.1 Icc M7 Profitability

In order to define the number of days making buying an ICC M7 server profitable, we divide the purchasing price by the renting price per day. The result is rounded to the nearest integer.

$$\frac{Purchasing_{IccM7}}{Renting_{IccM7}} = 1893 \tag{1}$$

### 0.2 E.2 Container and 4 Raspberry PI Profitability

To get the number of days of renting a container making the cost higher than buying and running 4 Raspberry Pis, we apply the following function:

If we rent containers working as 4 Raspberry Pis, the daily costs are:

$$\begin{aligned} Container_{DailyCost} &= 4 * (container\_cost_{per\_GB/day} * RPi\_RAM\_GB + \\ &\quad container\_cost_{per\_vCPU/day} * RPi\_TP\_vCPU) \\ &= 0.540864 \end{aligned} \tag{2}$$

If 4 Raspberry Pis are idle, the daily costs are:

$$\begin{aligned} 4RPi_{DailyCost\_Idle} &= 4 * RPi\_cost_{power\_idle} * time \\ &= 0.072 \end{aligned} \tag{3}$$

If 4 Raspberry Pis compute, the daily costs are:

$$\begin{aligned} 4RPi_{DailyCost\_Computing} &= 4 * RPi\_cost_{power\_computing} * time \\ &= 0.096 \end{aligned} \tag{4}$$

Minimum number of renting days for containers working as idle RPi is (rounded to the nearest integer):

$$\begin{aligned} Min\_days &= Price\_RPi/(Container_{DailyCost} - 4RPi_{DailyCost\_Idle}) \\ &= 926 \end{aligned} \tag{5}$$

Maximum number of renting days for containers working as computing RPi is (rounded to the nearest integer):

$$\begin{aligned} Max\_days &= Price\_RPi/(Container_{DailyCost} - 4RPi_{DailyCost\_Computing}) \\ &= 976 \end{aligned} \tag{6}$$

Therefore, the number of days of renting a container making the cost higher than buying and running 4 Raspberry Pis is between 926 and 976 days.

## 0.3 E.3 Icc M7 vs Raspberry Pis

For the same buying price as an ICC.M7, the number of Raspberry Pis we can get (floor the result to remove the decimal):

$$\begin{aligned} \#RPis\_Eq\_Buying\_ICCM7 &= (Price\_M7/Price\_RPi).floor \\ &= 355 \end{aligned} \tag{7}$$

So for the same buying price as an ICC.M7, we can get 355 Raspberry Pis and obtain a larger overall throughput and RAM. The throughput and RAM ratio are calculated in the following functions (the results are rounded to the fourth decimal place):

$$\begin{aligned} Throughput\_ratio &= \frac{\#RPis\_Eq\_Buying\_ICCM7 * RPi_RAM_GB}{M7_RAM_GB} \\ &= 1.8490 \end{aligned} \tag{8}$$

$$\begin{aligned} RAM\_ratio &= \frac{\#RPis\_Eq\_Buying\_ICCM7 * RPi_TP_vCPU}{M7_TP_vCPU} \\ &= 3.1696 \end{aligned} \tag{9}$$