CS-449 Project Milestone 2: Optimizing, Scaling, and Economics

Motivation and Outline: Anne-Marie Kermarrec
Detailed Design, Writing, Tests: Erick Lavoie
Teaching Assistant: Mu Zhou
Last Updated: 2022/04/25 11:35:03 +02'00'

Due Date: 20-05-2022 23:59 CET
Submission URL:
`https://cs449-submissions.epfl.ch:8083/login?next=%2Fm2`
General Questions on Moodle
Personal Questions: mu.zhou@epfl.ch

Abstract

In this milestone, you will parallelize the computation of similarities by leveraging the Breeze linear algebra library for Scala, effectively using more efficient low-level routines. You will also measure how well your Spark implementation scales when adding more executors. You will also implement a version of approximate k-NN that could be useful in very large datasets. You will finally compute economic ratios to help you choose the most appropriate infrastructure for your needs.

1 Motivation: Growing Pains

Your movie recommendation service is growing in number of users and movies rated, beyond the hobby side project you had started with. As your service is growing, the time to compute the *k-nearest neighbours* is growing also. If left unchecked, this may exceed the time your users are willing to wait to obtain high-quality suggestions.

In this last Milestone you are going to keep computation time within reasonable bounds in two steps: (1) you will first optimize your implementation for a single executor, with potential to lower the computation time to less than 3s, which might be a more than a 10x improvement compared to your implementation of Milestone 1 (you will do so in Section 3); (2) you will parallelize the execution of both the *k-NN* implementation and the predictions with Spark, using simple parallelization methods (Section 4). The optimization will keep your technical requirements lower, and the scaling will enable you follow the

growth of your users easily. You will also explore the potential for approximate k-NN methods (Section 5).

As your service is growing, so does the cost of the infrastructure you are using. As a practice for anticipating the costs of future infrastructure, and choosing the least costly options, you will compute a few simple economic ratios to compare some currently available options for the supporting hardware infrastructure (Section 6).

After completing this Milestone and the project, you will have had a basic but complete experience in implementing useful data analytics algorithms, measuring and analyzing their computing time, optimizing their performance both on a single machine and in a distributed fashion, and planning required resources for future growth. You should therefore have a good foundation for pursuing a research career in distributed systems, doing more advanced development on real-world infrastructure, as well as leading a technical team in a startup.

2 Dataset

For the optimization part of this milestone, you will again use the MovieLens 100K dataset [4]. Again, for the sake of simplicity, you will only test on the `ml-100k/u2.test` dataset (with the corresponding `ml-100k/u2.base`). This will enable you to evaluate performance gains compared to your implementation of Milestone 2.

For the scaling part of this milestone, you will use the 1M dataset, which should be more challenging than the 100K but still has a reasonable computation time. The original dataset can be found here:

`https://grouplens.org/datasets/movielens/1m/`

However, the 1M dataset does not provide pre-split training and test sets, as the 100K, so the scripts of the 10M dataset are used to split the ratings into a `ml-1m/rb.train` and `ml-1m/rb.test` datasets. For your convenience, the pre-split dataset has been added to the zip archive `data-m2.zip` here:

`https://gitlab.epfl.ch/sacs/cs-449-sds-public/project/dataset`

As for the 100K dataset, in a real-life setting you would test on multiple splits of the dataset to ensure your results are not specific to one split and most likely generalize. But for the sake of the pedagogical exercise, it will be sufficient to only test on the `ml-1m/rb.train` and `ml-1m/rb.test` split.

The minimum rating value of `ml-100k` and `ml-1m` is 1.0 in both cases, so no need to worry about ratings having a value of 0.5 *as in Milestone 1.*

3 Optimizing with Breeze, a Linear Algebra Library

Before throwing more computing resources at the problem, e.g. Spark Executors running on additional cluster machines, it is always worth verifying whether a single machine would actually be sufficient for the task. In some cases, as is the case for your recommender, a drastic reduction in computation time can be achieved with slightly different, but mathematically equivalent, implementation techniques.

As far as we are aware of, Scala does not offer complete, efficient, and mature linear algebra libraries. The Breeze library[1] does however come somewhat close on a subset of operations and datatypes. Breeze is the recommended interface for Scala/Spark to the underlying netlib-java library[2], itself a wrapper for low-level linear algebra libraries such as BLAS and LAPACK, that is used internally by the Spark mllib library. In this section, you will therefore use Breeze directly to reimplement the prediction algorithm of Milestone 1 and measure the speedup you can obtain. With relatively simple implementation techniques, you should be able to bring the running time within 3 seconds. Depending on how you implemented the previous Milestone, this might potentially offer a 10x reduction in computation time. Such a reduction can directly translate in a $10x$ reduction in the number of executors required to obtain the same computation time, potentially more depending on the overhead of RDD libraries and their parallelization behaviour.

Note that to best leverage a linear algebra library, your implementation should compute *all* k-nearest neighbours prior to making predictions, instead of lazily based on which predictions to make, as the RDD API might have encouraged you to do previously. You will notice that the speed up you obtain this way stays quite large even if you are actually computing more similarities than previously.

Note also that, although teaching additional programming environments is outside of the scope of the course, the techniques introduced in this section provide even better performance gains when used in Numpy (with Python) due to more complete and mature numerical libraries, and could also be integrated in Spark. This could be an additional valid option for your future projects.

3.1 Overview of some Breeze Operations

Unfortunately, the breeze library has been on minimal maintenance mode for a couple of years[3] and the documentation is somewhat lacking[4]. To get you

[1] `https://github.com/scalanlp/breeze`

[2] `https://github.com/fommil/netlib-java`

[3] `https://github.com/scalanlp/breeze/commits/master`

[4] This is no fault of the author, as the library does quite a lot as a labor of (volunteer) love, and as such is already quite an accomplishment. This highlights the issue of funding to maintain core open source infrastructure. If you ever work for a company that makes money using open source software, do remember to pay the benefits forward and sustain the open

started, here is an overview of some operations we found useful in our reference implementation.

For all of them you should first import:

```
import breeze.linalg._
import breeze.numerics._
```

3.1.1 Dense Vector and Matrix

If all the data can fit in memory, the most efficient numerical operations are usually based around the `DenseVector` and `DenseMatrix` data types. There is a fairly comprehensive library for linear algebra, which supports efficient slicing, element-wise operations, and matrix multiplication. The set of operations and their equivalence in other numerical languages is listed here: `https://github.com/scalanlp/breeze/wiki/Linear-Algebra-Cheat-Sheet`.

When working with the `ml-100k/u2.base` dataset, a `DenseMatrix` can easily fit all the ratings in memory, even if they are sparse compared to $users * items$. Prefer a sparse matrix anyway since it makes it easier to try larger datasets with the same implementation, as you will have to do in the next question.

3.1.2 Sparse Matrix

If a matrix, e.g. user-item ratings, does not fit in memory using a `DenseMatrix`, as is necessarily the case for the largest MovieLens datasets, you can use a `SparseMatrix` representation. The Breeze library only supports the Compressed Sparse Column (CSC) format[5]. Here is a summary of operations you might find useful.

Creation

You can efficiently create a `CSCMatrix` with the `CSCMatrix.Builder` as follows:

```
val builder = new CSCMatrix.Builder[Double](rows=943, cols=1682)
builder.add(row, col, value)
...
val x = builder.result()
```

The template already does this for the ratings, both for training and testing.

Slicing

You can obtain a `SliceVector` of rows i to j of column m with `x(i to j, m)` or a `SliceMatrix` of row i to j of columns m to n with `x(i to j, m to n)`. Note that if you slice to obtain a single row i between columns m and n you will obtain a "transposed" vector, which you may have to transpose again

source commons by funnelling some funding towards the projects you depend on.

[5]See Wikipedia for an overview of other formats: `https://en.wikipedia.org/wiki/Sparse_matrix`

for some operations as they may not support the transposed version. Note also that the `::` operator (ex: `x(i, ::)`), which provides all elements along the corresponding dimension, is not supported on `SparseMatrix`. You have to explicitly provide a range `i to j` for that dimension.

Iteration

You can efficiently iterate through all non-zero elements of a `CSCMatrix` matrix x with:

```
for ((k,v) <- x.activeIterator) {
  val row = k._1
  val col = k._2
  // Do something with row, col, v
}
```

Matrix multiplication

You can multiply matrix `x` by matrix `y` (both can also be slices with compatible dimensions) with `x * y`. You can use it to implement the $s_{u,v}$ computation with pre-processed ratings of Milestone 2:

$$s_{u,v} = \sum_{i \in (I(u) \cap I(v))} \breve{r}_{u,i} * \breve{r}_{v,i} \tag{1}$$

Matrix-Vector multiplication

As an alternative to matrix multiplication, you can also do a series of matrix-vector multiplication, extracting individual rows or columns of another matrix. When using this approach, you have to figure out if it is best to assign users to rows and items to columns of a `CSCMatrix` (or the opposite!) when preparing the $\breve{r}_{u,i}$ sparse matrix.

Reduction

The reduction along a given dimension, such as performing a `sum(x, Axis._1)` is not supported. You can either explicitly create a `DenseVector` with the correct size for that dimension then iterate through all active (non-zero) elements as shown above. You can also do a matrix multiplication: for a matrix X, you can reduce the columns by multiplying $X_{m,n} * 1_{n,1}$ or the rows by multiplying $1_{1,m} * X_{m,n}$, where $1_{1,n}$ is an `n-by-1` matrix of all 1 and $1_{1,m}$ is a `1-by-m` matrix of all 1. Either options are useful for pre-processing the ratings, i.e. computing $\breve{r}_{u,i}$, prior to computing $s_{u,v}$, you will have to measure which one is fastest.

top-k elements

You can find the indices of the top `k` elements of a (non-transposed) `SliceVector` `s` with `argtopk(s, k)` and iterate through them with:

```
for (i <- argtopk(s,k)) {
```

```
  // do something with s(i)
}
```

It can sometimes be faster to call `argtopk` with a `DenseVector` instead of a `SliceVector`. You will have to measure whether this is significant.

Element-wise operations

Many element-wise operations and boolean operations from `https://github.com/scalanlp/breeze/wiki/Linear-Algebra-Cheat-Sheet` are supported on `CSCMatrix`. Unsupported operations will result in compilation errors. You can quickly and interactively identify and test those that are supported with `sbt console` on a small test `CSCMatrix`, which you may create as such:

```
> val x = CSCMatrix[Double]((1.0, 2.0, 3.0), (4.0, 5.0, 0.0))
// x.<tab> for autocompletion of supported operations
```

Others

See `https://github.com/scalanlp/breeze/blob/06b23cfa837c53e025bb70f0f4bc1f241986d0ba/math/src/test/scala/breeze/linalg/CSCMatrixTest.scala` for example usage of other supported operations on `CSCMatrix`.

3.2 Questions

BR.1 Reimplement the kNN predictor of Milestone 1 using the Breeze library and without using Spark. Using $k = 10$ and `data/ml-100k/u2.base` for training, output the similarities between: (1) user 1 and itself; (2) user 1 and user 864; (3) user 1 and user 886. Still using $k = 10$, output the prediction for user 1 and item 1 ($p_{1,1}$), the prediction for user 327 and item 2 ($p_{327,2}$), and make sure that you obtain an MAE of 0.8287 ± 0.0001 on `data/ml-100k/u2.test`.

BR.2 Try making your implementation as fast as possible, both for computing all k-nearest neighbours and for computing the predictions and MAE on a test set. Your implementation should be based around `CSCMatrix`, but may involve conversions for individual operations. We will test your implementation on a secret test set. The teams with both a correct answer and the shortest time will receive more points.

Using $k = 300$, compare the time for predicting all values and computing the MAE of `ml-100k/u2.test` to the one you obtained in Milestone 1. What is the speedup of your new implementation (as a ratio of $\frac{average\ time_{old}}{average\ time_{new}}$)? Use the same machine to measure the time for both versions and provide the answer in your report.

Also ensure your implementation works with `data/ml-1m/rb.train` and `data/ml-1m/rb.test` since you will reuse it in the next questions.

3.3 Tips

1. User and movie indexes start at 1 in the movielens dataset, but the `CSCMatrix` implementation is 0-indexed. If you map user 1 to index 0, make sure that all your operations take into account the difference.

2. Idem if you decide to map users to columns instead of rows of the `CSCMatrix`.

3. For any kind of optimization work, always make sure to first have a correct (and often simpler) implementation to compare against, to ensure your optimizations do not affect the correctness of the results. As you develop, compare the answers for both and ensure their answers stay the same. Your implementation of Milestone 1 can serve for this.

4. Prior to optimizing, make sure to measure how long each part of your program takes. Our intuitions of what is fast and slow is (most) often wrong, so always measure first to focus you attention on the parts that matter most.

5. Always tackle the slowest parts first, then move to the others. As you optimize, the bottlenecks, i.e. the parts that slow your program the most, will change.

6. Optimization work can be addictive. It also suffers from diminishing returns, i.e. the biggest gains can be obtained with relatively little work but further gains will take increasingly larger amounts of work. For the sake of the project, we are aiming for a $10x$ reduction in computation time for both the *k-NN* computations and the prediction time. If your implementation of Milestone 1 was already quite fast, you may not easily obtain a 10x reduction, that is fine.

4 Parallel k-NN Computations with Replicated Ratings

In a real-world setting, your number of users may outgrow the capabilities of a single machine, even after extensive optimization work. In that case, distributing the execution on multiple machines allows your implementation to grow with the number of users. We will focus on parallelizing *k-NN* and predictions.

You will first use a simple parallelization strategy in which every worker (Spark executor) will obtain a copy of the input data and compute different parts of the output. The distribution of the shared input data is done efficiently with Spark's *broadcast variables*.

Workers will compute the top k users for different subsets of users. The results are then reassembled on the driver node in a sparse *k-NN* `CSCMatrix`. A high-level presentation of the parallelization strategy is given in Algorithm 1.

Algorithm 1 Parallel k-NN Computations with Replicated Ratings (Spark)

Input $r_{\bullet,\bullet}$ (ratings), sc (SparkContext), k (number of nearest neighbours)
$\breve{r}_{\bullet,\bullet} \leftarrow$ preprocess($r_{\bullet,\bullet}$) ▷ Similar to Milestone 2
$br \leftarrow sc$.broadcast($\breve{r}_{\bullet,\bullet}$)
procedure $topk(u)$
    $\breve{r}_{\bullet,\bullet} \leftarrow br$.value ▷ Retrieve broadcast variable
    $s_{u,\bullet} \leftarrow sim(u, \breve{r}_{\bullet,\bullet})$ ▷ Similarities
    return $(u, argtopk(s_{u,\bullet}, k).map(v \rightarrow (v, s_{u,v})))$ ▷ Compute k-NN for user u
end procedure
$topks \leftarrow sc.parallelize(0$ **to** $nb\_users).map(topk).collect()$
$knn \leftarrow knnBuilder(topks)$ ▷ Using the CSCMatrix.Builder
return knn

4.1 Questions

In the following questions, you will ensure this distributed version gives the same results you obtained previously and assess its scalability and resource usage.

EK.1 Test your parallel implementation of k-NN for correctness with two workers. Using $k = 10$ and `data/ml-100k/u2.base` for training, output the similarities between: (1) user 1 and itself; (2) user 1 and user 864; (3) user 1 and user 886. Still using $k = 10$, output the prediction for user 1 and item 1 ($p_{1,1}$), the prediction for user 327 and item 2 ($p_{327,2}$), and make sure that you obtain an MAE of 0.8287 ± 0.0001 on `data/ml-100k/u2.test`

EK.2 Measure and report the combined *k-NN* and *prediction* time when using 1, 2, 4 workers, $k = 300$, and `ml-1m/rb.train` for training and `ml-1m/rb.test` for test, on the cluster (or a machine with at least 4 physical cores). Perform 3 measurements for each experiment and report the average and standard-deviation total time, including training, making predictions, and computing the MAE. 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? Answer both questions in your report.

4.2 Tips

1. Use the following command to vary the number of executors on the IC-Cluster (once connected on the `iccluster028.iccluster.epfl.ch` driver node): `spark-submit --master yarn --num-executors X --conf "spark.dynamicAllocation.enabled=false" ...` where `X` is the number of executors to use.

5 Distributed Approximate k-NN

Up to now, we have aimed at obtaining the same results in a distributed implementation of k-NN as for a centralized. For some applications, obtaining the k *best*, i.e. most similar, neighbours for every user is not necessary. As long as the k found are similar enough, this is sufficient. This relaxation enables the user ratings to be partitioned among multiple workers, with possible repetitions, and a *divide-and-conquer* approach to be used: local k-NNs are computed within each partition, ignoring users from other partitions, and the most similar across partitions will be kept. This approach increases the total number of ratings that can be held in memory, by only storing a subset on each worker, and decreases the time required for computing the k-NN, by avoiding most similarity computations.

However, the quality of the final approximate k-NN, i.e. how similar the k nearest neighbours are to any user, is greatly impacted by the initial partitioning of the ratings. For example, if users u and v, who are most similar to one another than all other users, end up in different partitions, they won't be considered for the k-NN and less similar neighbours will be used instead. It is therefore important to quantify the similarity lost by approximation, and its impact on predictions, to ensure the personalized recommender still provides better predictions than if a simpler and faster method had been used. In the next questions, you will implement and evaluate a simple random partitioning scheme[6] and quantify such tradeoffs.

5.1 Questions

AK.1 Implement the approximate k-NN using your previous breeze implementation and Spark's RDDs. Using the partitioner of the template with 10 partitions and 2 replications, $k = 10$, and `data/ml-100k/u2.base` for training, output the similarities of the approximate k-NN between user 1 and the following users: $1, 864, 344, 16, 334, 2$.

AK.2 Vary the number of partitions in which a given user appears. For the `data/ml-100k/u2.base` training set, partitioned equally between 10 workers, report the relationship between the level of replication (1,2,3,4,6,8) and the MAE you obtain on the `data/ml-100k/u2.test` test set. What is the minimum level of replication such that the MAE is still lower than the baseline predictor of Milestone 1 (MAE of 0.7604), when using $k = 300$? Does this reduce the number of similarity computations compared to an exact k-NN? What is the ratio? Answer both questions in your report.

AK.3 Measure and report the time required by your approximate *k-NN* implementation, including both training on `data/ml-1m/rb.train` and computing the MAE on the test set `data/ml-1m/rb.test`, using $k = 300$ on

[6] More sophisticated methods, such as Cluster-and-Conquer [3], can provide good partitioning at reasonable computation costs. If you are interested in delving in such methods, ask for a semester project with the SaCS lab.

8 partitions with a replication factor of 1 when using 1, 2, 4 workers. Perform each experiment 3 times and report the average and standard-deviation. Do you observe a speedup compared to the parallel (exact) k-NN with replicated ratings for the same number of workers?

6 Economics

You have seen that the optimizations of Section 3 may translate in lower needs for additional hardware. In this section, you will quantify the economics of buying/renting hardware to execute your recommender. You will, in the process, obtain insights into the economic benefits of optimization.

These days, you have many choices of hardware infrastructure. The landscape and particular tradeoffs of each point, are too broad to cover in this project. But since it may affect some design decisions, let's quickly review three representative examples of currently available hardware, summarized in Table 1, and provide some context in which they may be useful.

Hardware	**RAM**	**CPU**	**Buying Costs**	**Renting/Operating Costs**
ICC.M7	24x64GB DDR4 -2666	2x Intel Xeon Gold 6132 (Skylake) 2x 14 cores, 2.6 GHz, 19.25 MB L3 cache	(none)	(renting) 20.40CHF/day
Containers	Unspec.	Unspec.	(none)	(renting) 1 GB-s: $1.6e^{-7}$CHF/s 1 vCPU-s: $1.14e^{-6}$CHF/s
RPi 4 Compute Module	8GB LPDDR4 -3200 SDRAM	Broadcom BCM2711 quad-core Cortex-A72 ARMv8 64-bit SoC@1.5GHz	108.48CHF[7]	Power: 3W (idle) - 4W (computing)[8] Energy cost: 0.25CHF/kWh[9] + Maintenance

Table 1: Technical specifications and cost of IC IT cloud offerings, compared to the highest performing Raspberry Pi available in 2021, a cheap and widely available consumer computing device.

First, you may choose not to distribute your similarity computations and instead replace your re-purposed desktop machine, with a more powerful one, i.e. with increased RAM and better processor performance. One representative example is a powerful server. For example, the server in the IC IT Cluster with the largest amount of main memory, at 1.5TB of RAM, is listed in Table 1. The main advantage of this approach is that you don't have to change your software implementation, which sometimes is not possible because your particular algorithms are not easily distributed, and other times might be too complex for the skills or development time you currently have. However, the main drawback is that higher-end machines are usually more expensive because they are produced in lower volumes and sold at higher margins. The running costs at idle are also higher because all the added circuitry needs to be powered, even if not used.

Second, if your algorithms can be distributed, which fortunately is the case for k-NN [6, 5], you may elect to distribute them on virtualized cloud infrastructure. Today's cloud providers have many virtualization offerings, with one example based on containers, i.e. isolated processes running within the same operating system. Virtualization makes available the underlying hardware at a finer granularity, e.g. per CPU per second, as listed in the second line of Table 1. The main advantages of virtualized cloud offerings are potentially reduced maintenance costs, as the maintenance of the hardware is centralized and shared between multiple cloud users, and offloading the risks associated with under- or over-provisioning hardware infrastructure compared to actual business needs (you only pay what you use). However, if the actual maintenance costs for some applications are low, and computing needs are stable and foreseeable, the cost of acquiring hardware may actually be recouped in a few years. Moreover, if user data is subject to stringent privacy limitations, it may sometimes not be possible to process it in a cloud (although there are plenty of ongoing research and industrial developments aiming to provide better privacy guarantees). Also, the electronic waste, carbon emissions, and energy usage, of cloud providers may be higher than what you could obtain with privately-operated infrastructure [2].

Third, you may distribute your algorithms on hardware you acquire and maintain yourself, by favouring cheap, energy-efficient, and widely available commodity hardware, an approach that was pioneered by Google in 1998. For example, the highest performing Raspberry Pis are increasingly closing the gap with Desktop and Server performance at a really low price point. The specification and costs, as of March 2022, are listed in the third line of Table 1. While there is still a significant gap in performance with the most powerful hardware of today, RaspberryPi 4 Compute Modules are on par or more powerful than the servers used by Google in 2005, are more energy efficient by a factor of 4-5x, and probably less expensive by a factor at least 4-5x [1]! They provide the same advantages as Hardware-as-a-Service, in having full access to the hardware and

[7] From EUR to CHF, as listed here (Mar. 10th 2022) `https://buyzero.de/products/compute-module-4-cm4?variant=32090358612070&src=raspberrypi`

[8] `https://www.epfl.ch/labs/sacs/wp-content/uploads/2022/03/andre_bachelor_thesis_final.pdf`

[9] Swissgrid Tariffs (2022)

their operating costs (when only taking electricity into account) are lower than equivalent virtualized offerings for the same performance. Moreover, private data never leaves the organization, the hardware can be used for longer than the typical 2-3 years turn-around time of cloud servers (lowering e-waste and grey energy), and they can be setup to operate on renewable energy at reasonable costs [2]. However, setting up the infrastructure and maintaining the software up-to-date requires labor time, which may more than offset the lower operating costs. They also have higher initial costs compared to cloud offerings, but, given the low price of individual devices, that cost may be spread over time and engaged as computing needs grow.

The following questions will help you acquire or rent the right amount of hardware resources for your current and future needs.

6.1 Questions

For the following questions, make the following assumptions[10]:

- Throughput of 1 single Intel CPU physical core (vCPU) $\approx$ 4 Raspberry Pis 4, (16 Cortex-A72 cores total) [11]
- Buying a machine equivalent to the ICC.M7 costs 38,600 CHF (based on a 41,500$USD quote from the Dell website in the US, created on Mar. 10th 2022)

Compute your answers to the following questions with code (see the template):

E.1 What is the minimum number of days of renting to make buying the ICC.M7 less expensive, excluding any operating costs such as electricity and maintenance? Round up to the nearest integer.

E.2 After how many days of renting a container, is the cost higher than buying and running 4 Raspberry Pis? (1) Assuming optimistically no maintenance at minimum power usage for RPis, and (2) no maintenance at maximum power usage for RPis, to obtain a likely range. (Round up to the nearest integer in each case).

Assume a single processor for the container and an equivalent amount of total RAM as the 4 Raspberry Pis. Also provide unrounded intermediary results for (1) Container Daily Cost, (2) 4 RPis (Idle) Daily Electricity Cost, (3) 4 RPis (Computing) Daily Electricity Cost.

[10] For real deployments, you would have to empirically verify these assumptions, and revise them as the application, the landscape of hardware offerings, and your software implementation evolve.

[11] Rough estimate based on the performance of the more recent Intel Xeon E3-1230 v6, which has 4 cores instead of 14, on the following benchmark: `https://truebench.the-toffee-project.org/`. Note that the Intel cores have hyper-threading, which doubles some of hardware to obtain twice the number of virtual CPUs and is clocked almost twice higher, so the actual performance per ARM core is actually much better than it may seem at first sight.

E.3 For the same buying price as an ICC.M7, how many Raspberry Pis can you get (floor the result to remove the decimal)? Assuming perfect scaling, would you obtain a larger overall throughput and RAM from these? If so, by how much? Compute the ratios using the previous floored number of RPis, but do not round the final results.

Implement the computations for the different answers in the Economics.scala file. You don't need to provide unit tests for this question, nor written answers for these questions in your report.

6.2 Tips

1. Write down the full equation with units for every term when calculating ratios.

7 Deliverables

You can start from the latest version of the template:

- `https://gitlab.epfl.ch/sacs/cs-449-sds-public/project/cs449-Template-M2-2022/`

We may update the template to clarify or simplify some aspects based on student feedback during the next weeks, so please refer back to see the latest changes.

Provide answers to the previous questions in a **pdf** report (if your document saves in any other format, export/print to a pdf for the submission). Also, provide your source code (in Scala) in a single archive:

```
report.pdf
src/*
```

Your archive should be around or less than 1MB. Submit to the TA using the submission URL of the first page of this Milestone.

8 Grading

You will be graded both on the question answers, and source code quality as well as organization. Points for source code will reflect how easy it was for the TA to run your code and check your answers. Grading for answers to the questions without accompanying executable code will be 0.

8.1 Collaboration vs Plagiarism

You are encouraged to help each other better understand the material of the course and the project by asking questions and sharing answers. You are also very much encouraged to help each other learn the Scala syntax, semantics, standard library, and idioms and Spark's Resilient Distributed Data types and APIs. It is also fine if you compare answers to the questions before submitting

your report and code for grading. The dynamics of peer learning can enable the entire class to go much further than each person could have gone individually, so it is very welcome.

However, each team should write their own original report and code. We will give 0 to all submissions that are copies of one another from this year, and also 0 to copies of previous years' submissions.

9 Updates

Since the original release of the Milestone description, we have made the following changes:

- None, yet!

References

[1] Google machine. `https://google-services.blogspot.com/2006/07/google-machine.html`, 2006. Accessed: 2021-01-21.

[2] De Decker, K. Low-tech magazine: Solar-powered website. `https://solar.lowtechmagazine.com/about.html`, 2019. Accessed: 2021-01-21.

[3] Giakkoupis, G., Kermarrec, A.-M., Ruas, O., and Taïani, F. Cluster-and-Conquer: When Randomness meets Graph Locality. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), IEEE, pp. 2027–2032.

[4] Harper, F. M., and Konstan, J. A. The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems 5*, 4 (Dec. 2015), 19:1–19:19.

[5] Karydi, E., and Margaritis, K. Parallel and distributed collaborative filtering: A survey. *ACM Comput. Surv. 49*, 2 (Aug. 2016).

[6] Schelter, S., Boden, C., and Markl, V. Scalable similarity-based neighborhood methods with mapreduce. In *Proceedings of the Sixth ACM Conference on Recommender Systems* (New York, NY, USA, 2012), RecSys '12, Association for Computing Machinery, p. 163–170.

A Notation

- u and v: *users* identifiers
- i and j: *items* identifiers
- $\bar{r}_{\bullet,\bullet}$: average over a range, with $\bullet$ representing all possible identifiers, either *users*, *items*, or both (ex: $\bar{r}_{\bullet,i}, \bar{r}_{u,\bullet}, \bar{r}_{\bullet,\bullet}$), ex: $\bar{r}_{u,\bullet} = \frac{\sum_{r_{u,i} \in \text{Train}} r_{u,i}}{\sum_{r_{u,i} \in \text{Train}} 1}$

- $\hat{r}_{u,i}$: deviation from the average $\bar{r}_{u,\bullet}$
- $r_{u,i}$: rating of user u on item i, (u is always written before i)
- $p_{u,i}$: predicted rating of user u on item i
- $|X|$: number of items in set X
- $*$: scalar multiplication
- $r_{u,i}, r_{v,i} \in$ Train: both $r_{u,i}$ and $r_{v,i}$ are elements of $Train$ for the same i
- 1_x: indicator function, $\begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{otherwise} \end{cases}$
- $u, v \in U$: shorthand for $\forall u \in U, \forall v \in U$
- $R(u, n)$: top n recommendations for user u as a list
- $sorted_{\searrow}(x)$: sort the list x in decreasing order
- $[x|y]$: create a list with elements x such that y is true for each of them
- $top(n, l)$: return the highest n elements of list l
- U: set of users
- I: set of items
- $U(i)$: is the set of users with a rating for item i ($\{u|r_{u,i} \in \text{Train}\}$)
- $I(u)$: is the set of items for which user u has a rating ($\{i|r_{u,i} \in \text{Train}\}$)

B Pedagogical Notes

In this appendix, we explain why we structured the Milestones as they are. This is not necessary to complete the Milestone successfully but might be of interest for future revisions and curious students. Among other goals, the entire project aims to:

- Develop the skills required to implement and optimize variations of a data science algorithms locally and in a distributed manner;
- Understand how k-NN fits in a recommender application, and how it relates in performance and accuracy to other similar methods;
- Understand the methodology to build reliable distributed applications in a step-wise fashion;
- Compare the benefits and costs of distributed and non-distributed approaches to find where each is most applicable;
- Develop the skills to choose the most affordable infrastructure for different operation contexts.

Moreover, for the previous sections, here are specific goals:

B.1 Optimizing with Breeze

Progression : Optimize k-NN locally on a single machine

Skills : Leverage hardware parallelism and faster vectorized instructions through a linear algebra library

Methodology : Measure speedups

Knowledge : Reformulate parallel operations as matrix algebra

Evaluate : Programming skills for fast code with Breeze

B.2 Parallel Exact k-NN

Progression : Parallelize k-NN

Skills : Implement parallel k-NN with Spark

Methodology : Measure speedups

Knowledge : Impact of increasing the number of workers

Evaluate : Programming skills with Spark

B.3 Distributed Approximate k-NN

Progression : Relax accuracy for potential further performance gains

Skills : Implement approximate k-NN with Spark

Knowledge : Impact of the number of partitions and replication factor on accuracy

Evaluate : Programming skills with Spark

B.4 Economics

Progression : Step back from implementation, and consider the wider economic execution context

Skills : Compute ratios to decide when renting or owning is preferable

Knowledge : Compare the relative benefits and costs of widely available consumer devices to other alternatives

Evaluate : Computing of simple ratios