Big Data course project
Spark recommendation system: performance analysis.

University of Verona

Matteo Marjanovic

June 2023

1 Introduction

For the Big Data course project I wanted to explore the potential of Apache Spark not only in data loading, extraction and transformation. In fact, I wanted to use the Spark MLlib library to create a recommendation system. This because I was curious to know some use cases that need to process big data with tools like Spark and because during my master studies I attended many courses that let me approach with machine learning, a field that I'm really interested in. By the way, these courses were focused mainly on computer vision tasks, so I've the Big Data course as an opportunity to explore another field of application of machine learning: recommendation systems.

For this project I took inspiration from a Medium article written by Lijo Abraham, which cn be found at this link. To make this project more personal I tried to adapt his project to another dataset to see if I'd been able to achieve similar results. After that, I tried to use different subset of the dataset, each one of different dimension, to make a performance analysis of that approach.

2 Materials and Methods

2.1 Datasets

The datasets used for this projects have been two: the *Amazon product data* dataset (which is the one used in the article of Lijo Abraham), available here, and the *MovieLens* dataset, downloadable at this link.

2.1.1 Amazon product data

The *Amazon product data* dataset, as the name implies, is a collection of data that come from the famous amazon.com online shopping website. This is made of two parts: the first is *products metadata*, which contains products' metadata (product ID, title, price, brand, categories, description), and the second is *rating data*, which contains the ratings of the products left by the users in a 1 to 5 scale, (user ID, product ID, rating). Both of them are contained in a *.json.gz* file.

For what concerns this dataset, it's worth noting that the *categories* field of the products' metadata is an array of three elements created in a hierarchical way: this means that if our product is, for example, some running shoes of a certain brand, the value of the *category* field would be `["Sports & Outdoors", "Running", "Running Shoes"]`. In this project I considered only the "lower level" category that, in the example case, would be `Running Shoes`.

2.1.2 MovieLens

The *MovieLens* dataset is the result of the collection of movie ratings data from the movielens.com website made by GroupLens Research Lab. This website consist in a service which provide non-commercial personalized movie recommendations based on the ratings that users upload on it. This dataset, as the previous one, is made by two main parts: the *movies metadata* containing the movies' principal informations (movie ID, title, genres) and the *rating data*, containing the ratings left from the users of MovieLens about the movies, in a scale from 1 to 5 (user ID, movie ID, rating). Both of them are contained in a *.csv* file.

2.2 Spark and MLlib

Apache Spark™ is an open-source analytics engine designed for processing large-scale data. It offers a unified platform that enables programming clusters with implicit data parallelism and fault tolerance. In the context of big data, traditional computing frameworks may struggle to efficiently process and analyze large volumes of data due to limitations in scalability and computational power. Spark addresses these challenges by providing a distributed computing model that enables parallel processing across a cluster of machines. The foundational architecture of Apache Spark is built upon the concept of Resilient Distributed Datasets (RDDs). RDDs are fault-tolerant, read-only collections of data items distributed across a cluster of machines, ensuring data resilience and fault tolerance. Spark and its Resilient Distributed Datasets (RDDs) were created in 2012 as a response to the constraints of the MapReduce cluster computing model. MapReduce imposes a linear dataflow structure on distributed programs, where data is read from disk, mapped, reduced, and stored back on disk. In contrast, Spark's RDDs act as a distributed working set for programs, providing a deliberately restricted form of distributed shared RAM, which enables faster computations. In this work I used the Dataframe API, which was released as an abstraction on top of the RDD.

One of the advantages of using Spark is that wecan also rely on many components built upon it. It's the case of MLlib, a distributed machine learning framework built on top of Spark Core. Thanks to the distributed memory-based architecture of Spark, MLlib achieves significantly faster performance compared to the disk-based implementation used by Apache Mahout. Benchmarks conducted by MLlib developers, specifically on the alternating least squares (ALS) implementations, showed that MLlib outperformed Mahout by up to nine times. Additionally, MLlib exhibits superior scalability when compared to Vowpal

Wabbit. In this work I used the MLlib ALS implementation in order to perform collaborative filtering.

2.3 Elasticsearch

Elasticsearch, also known as ES, is a contemporary search and analytics engine that originated in 2010. It is built on the foundation of Apache Lucene and developed using Java. Elasticsearch falls under the category of NoSQL databases, which means it stores data in an unstructured manner and does not support querying with SQL.

In ES, an index is similar to a database in traditional systems: it is a collection of documents that are logically grouped together and share a similar structure. *Documents* are the basic units of data in Elasticsearch. They're represented as a JSON objects and contain a set of key-value pairs. Documents are stored within an index and can be retrieved, updated, or deleted individually. Finally, a *mapping* in ES defines the schema or structure of the documents within an index. It specifies the data types, field names, and other properties associated with the fields. Mapping allows Elasticsearch to understand and index the data efficiently.

In this project, ES is used to calculate cosine similarity between item factors obtained by ALS algorithm (1), querying on the items belonging to the same category as the chosen one.

2.4 Collaborative Filtering

A recommendation system, also known as a recommender system, is a machine learning approach that utilizes data to assist in predicting, filtering, and discovering relevant options for individuals amidst a vast and continuously expanding array of choices. These can be implemented mainly in two ways:

- **content-based systems** analyze the characteristics or properties of recommended items. For example, if a Netflix user has shown a preference for watching cowboy movies, the system would suggest movies from the database that are classified under the ”cowboy” genre.

- **collaborative filtering systems** make recommendations by measuring the similarity between users and/or items. These systems suggest items to a user based on the preferences of other users who have similar tastes or preferences.

Collaborative filtering is based upon a matrix which contains the user ratings for each item to suggest, that has users on rows and items on columns. This matrix is called **utility matrix**. To produce suggestions, the most common approach is to compute a row/column that is similar to the choosen one; this similarity is calculated using a certain metric as, for example, the cosine similarity.

In real world datasets, anyway, the vast majority of movies receive very few or even no ratings at all by users. Thus, if we look to the utility matrix, we find a very sparse matrix with most of the entries that are missing values. It's difficult to use the metrics cited previously in this conditions if we don't know how to treat missing values.

One possible solution is matrix factorization: in fact, we can factorize the matrix using two rectangular matrixes, one for the users and one for the items, trying to get one with similar values to what we have and, with that factors, fill all the missing values. There are many algorithms that are capable to do this operation of factorization, the one we used is called *alternating least squares*. ALS minimizes two loss functions alternatively: it first holds user matrix fixed and runs gradient descent with item matrix; then it holds item matrix fixed and runs gradient descent with user matrix (Algorithm 1). In our case, instead of reaching convergence (as in the shown algorithm) we train for five epochs, which allow as to reach a Mean Squared Error of ~ 0.2.

Algorithm 1 Alternating Least Squares (ALS)

```
Initialize user and item latent factor matrices randomly.
repeat
    Fix user latent factors and update item latent factors.
    for each item do
        Calculate weighted sum of user latent factors based on ratings.
        Update item latent factors using regularization and optimization
    end for
    Fix item latent factors and update user latent factors.
    for each user do
        Calculate weighted sum of item latent factors based on ratings.
        Update user latent factors using regularization and optimization
    end for
until convergence
```

In this project, I utilized the collaborative filtering technique within Pyspark to develop a recommendation system. Apache Spark MLlib provides an implementation of the widely adopted ALS algorithm for collaborative filtering, which is highly regarded for its effectiveness in generating recommendations. ALS runs its gradient descent in parallel across multiple partitions of the underlying training data from a cluster of machines.

One limitation of Spark ALS is that it primarily focuses on recommending the top products for a specific user (user-products model) or the top users for a specific product (product-users model). The computation of pairwise similarities for large product catalogues becomes challenging as the number of combinations grows exponentially ($O(n^2)$), resulting in prohibitively expensive shuffle operations and impractical computation times. To address this, I leveraged Spark and Elasticsearch to construct an item-item model. I utilized the the similarity of item factors obtained from the ALS model to efficiently calculate and store item-item similarities, enabling scalable and efficient recommendation computations.

In other words, the ALS algorithm factorize the utility matrix using two matrices, namely userFeatures and itemFeatures, to extract latent factors. With this approach, we compute the cosine similarity on the itemFeatures rank matrix (using ES) to determine item-item similarity. By comparing the vectors representing item features, we can identify items that exhibit similar characteristics or patterns, facilitating effective recommendation generation.

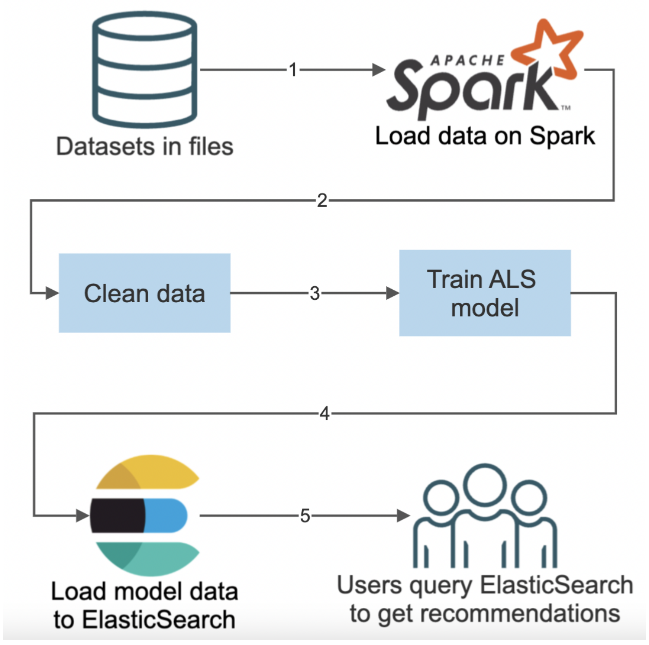
Datasets in files
1
APACHE Spark
Load data on Spark
2
Clean data
3
Train ALS model
4
5
Load model data to ElasticSearch
Users query ElasticSearch to get recommendations

Figure 1: Workflow of the recommendation system.

2.5 Workflow

Our project's workflow is composed by 5 steps (Figure 1):

1. **Load the product dataset into Spark**: read and infer the schema of the product dataset using Spark's data ingestion capabilities.

2. **Clean up the dataset using Spark DataFrame operations**: perform data cleaning operations, such as removing duplicates, discard useless columns and handling missing values. This operations serve to prepare the data to fit as input of the ALS model for collaborative filtering.

3. **Train a collaborative filtering model using Spark MLlib**: use Spark MLlib's collaborative filtering algorithm, such as ALS, to train a recommendation model from the rating data. Specify the necessary parameters like user and item columns, latent factors, and regularization.

4. **Load the cleaned data into Elasticsearch**: utilize the Elasticsearch connector for Apache Spark to save the cleaned DataFrames and the model data into an Elasticsearch index.

5. **Generate recommendations using Elasticsearch queries**: utilize Elasticsearch's search and query capabilities to generate recommendations based on item similarities using cosine distance.

3 Experiments

As said before, two datasets have been used. For **MovieLens** dataset I used three different subsets, given directly by the dataset's owners, which differs for the number of user ratings and products:

- **ml-latest-small**: ∼100'000 ratings, ∼9'000 movies;
- **ml-1m**: ∼1'000'000 ratings, ∼4'000 movies;
- **ml-10m**: ∼10'000'000 ratings, ∼10'000 movies;
- **ml-25m**: ∼25'000'000 ratings, ∼62'000 movies.

For what concerns **Amazon product data**, I selected one specific category, *CDs and Vinyl*, and divided the dataset in three groups:

- **amz-500k**: ∼500'000 ratings;
- **amz-1M**: ∼1'000'000 ratings;
- **amz-3M**: ∼3'000'000 ratings.

These are subsets given for this product category directly from dataset's owner. The number of products is ∼500'000 for all the three groups.

Since it is the Big Data course project, for the experiments I put the focus on Spark framework performance. The metrics I considered are the CPU usage, the RAM usage, shuffle read and shuffle write memory during the jobs.

In particular, shuffle read refers to the process of fetching data from the output partitions of preceding stages for redistribution to the tasks of subsequent stages and involves transferring data over the network from the nodes where the data was originally computed to the nodes where the data is required for further processing; shuffle write refers to the process of storing the data generated by a stage in such a way that it can be efficiently read by subsequent stages during the shuffle operation and involves partitioning and sorting the data based on the grouping criteria to prepare it for redistribution.

I have analyzed these metrics for the operations that I have considered to be the main ones in the creation of the recommendation model:

1. loading and cleaning (remove duplicates and null values) the movie/products data;
2. loading and cleaning (remove duplicates and null values) the user ratings data;
3. preparing the utility matrix to give as input to the ALS model;
4. training the ALS model;
5. evaluating the ALS model.

These steps are almost the same for both the datasets. The main difference is that the files containing data are *.csv* for MovieLens and *.json.gz* for Amazon product data. This means that to load them into a Spark DataFrame I used two different approaches: for csv files I used the PySpark built-in (and, consequently, optimized) method `spark.read.csv()`, for json.gz ones I used a custom function suggested by the dataset authors, which creates the DataFrame object parsing the file line by line. This influenced a lot the performances of the data loading stages.

One thing to mention is that I had to change the *Spark driver memory* parameter configuration (default 1 GB) due to the occurrence of "`java.lang.OutOfMemoryError: Java heap space spark`" error, due to low allocated memory problem.

The spark.driver.memory configuration in Apache Spark determines the memory allocated to the driver program, which runs on the Java Virtual Machine (JVM). It is important to set this configuration appropriately to avoid memory-related issues and ensure smooth execution. The driver program is responsible for coordinating the Spark application, executing tasks, and collecting results.

The changes have been the following:

- for MovieLens ml-25m subset, that parameter has been set to 3 GB since with the default value and with 2 GB the error occurred during the fifth phase (ALS model evaluation);
- for Amazon product data amz-1M subset, that parameter has been set to 2 GB since with the default value the error occurred during the fifth phase (ALS model evaluation);
- for Amazon product data amz-3M subset, that parameter has been set to 7 GB since with the default value, 2 GB, 3 GB and 4 GB the error occurred during the fourth phase (ALS model training), and with 5 GB and 6 GB the error occurred during the fifth phase (ALS model evaluation);

The CPU and RAM usage have been analyzed for each one of these steps, the shuffle read and write metrics that have been considered are the ones for the whole execution (the sum of the five steps).

One last aspect to mention before discussing the results is the architecture of the PC where these work have been executed. It has 16 GB of DDR4 RAM and a Intel Core i9-10900X 3,7 GHz CPU with 10 cores and 20 threads. For that reason, the spark RDD Blocks have been 20.

4 Results

4.1 Movies/products data loading and cleaning

Starting from the items data loading, the statistics are shown in Figure 2 for MovieLens and in Figure 3 for Amazon product data.

Looking at the MovieLens statistics we can see that for each of the four subset it's not been a job which used a lot of resources, it took about 4 seconds and used a low amount of CPU and RAM.

It's gone very differently for the Amazon product data. For all the three subsets the number of products is the same, so that's there are almost identical plots. What we can see is that it took around 40 seconds to load all the products. The first thing we could say is that they are 500'000, while the movies of the largest MovieLens subset are 62'000. But the main reasons for this difference are that the json.gz file is much larger than csv file (3MB vs 140MB) and, also,

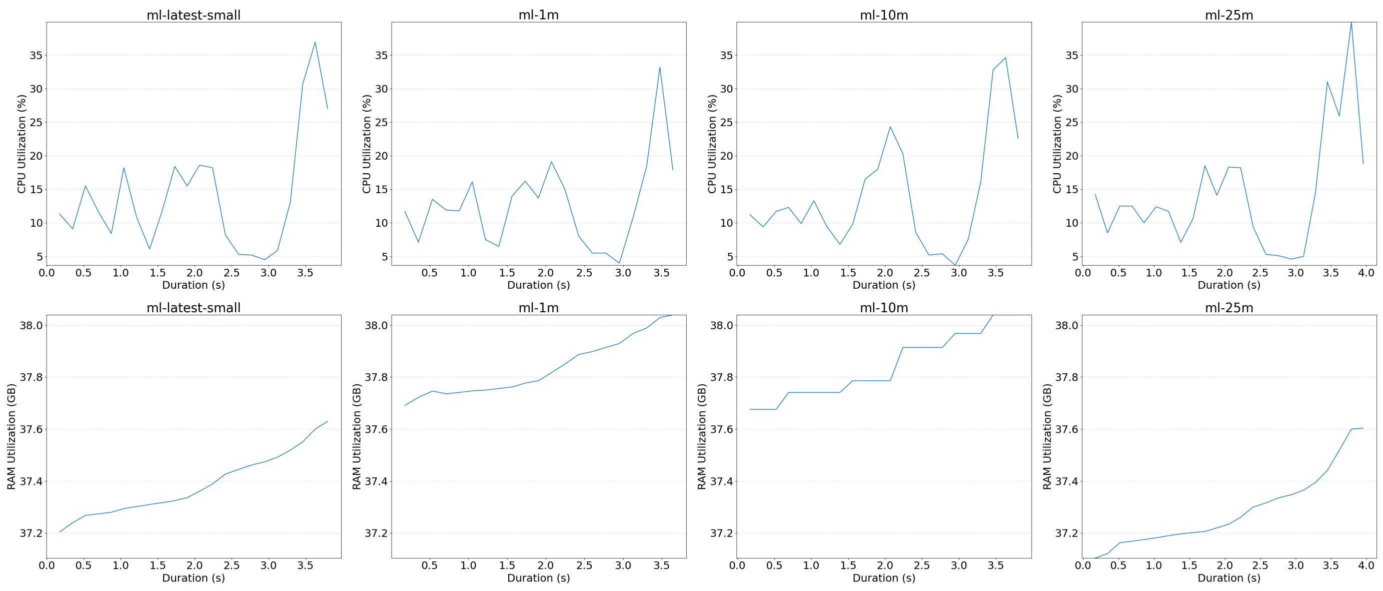

Figure 2: Statistics for first step (loading items data) for MovieLens dataset.

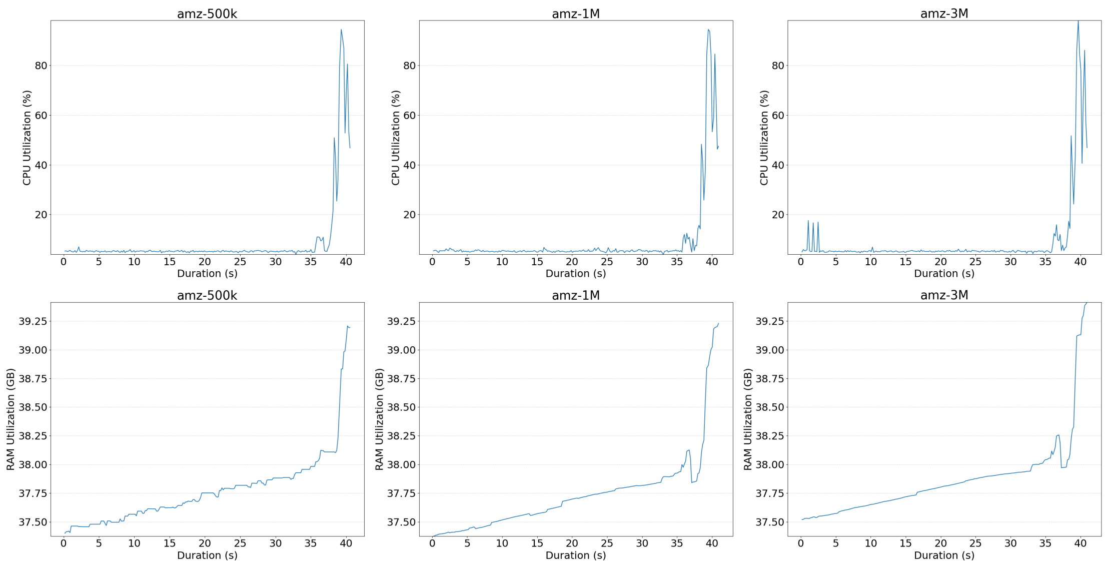

Figure 3: Statistics for first step (loading items data) for Amazon product data dataset.

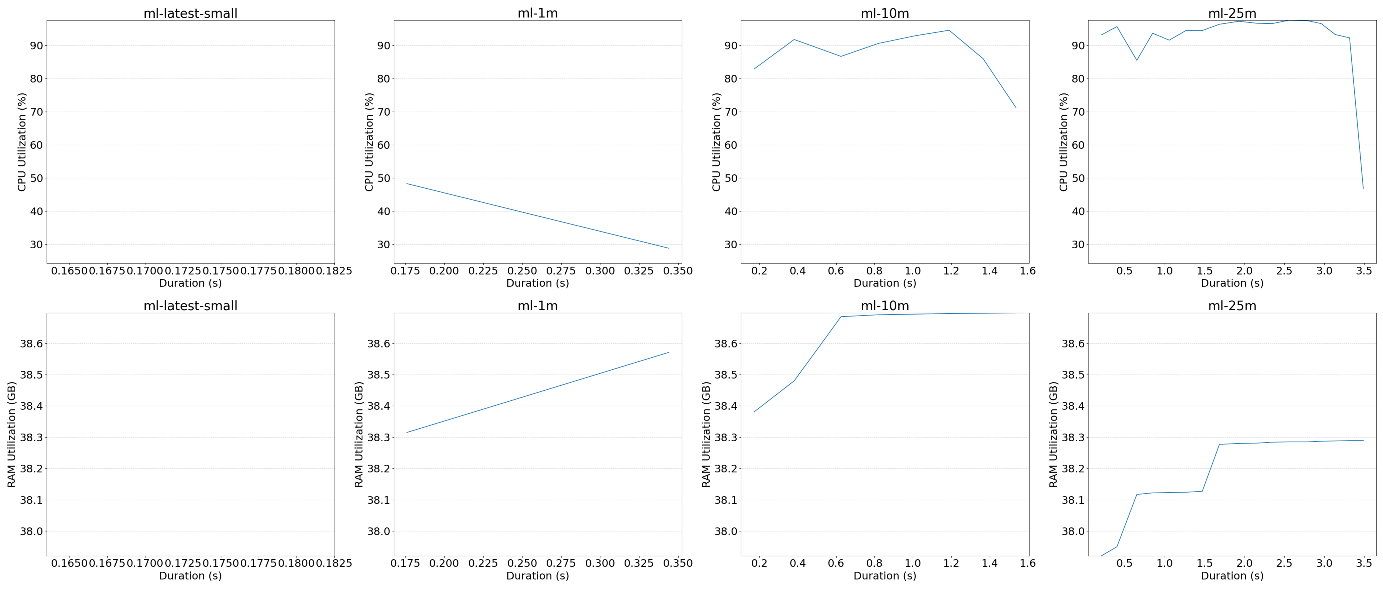

Figure 4: Statistics for second step (loading user ratings data) for MovieLens dataset.

the fact that the csv file is loaded with a Spark method which is optimized to do that and not with a custom method, as the json.gz file requires.

4.2 Ratings data loading and cleaning

For what concerns the loading and cleaning of the user ratings data, the statistics are in Figure 4 for MovieLens and in Figure 5 for Amazon product data.

For the MovieLens dataset we can see that the duration is never more than 3.5 seconds and the RAM is not stressed. Looking at the CPU usage we can see a clear difference between the first two and the least two subsets: in the first the usage is always under 50%, in the least it's constantly over 80%.

For the Amazon product data dataset the duration is a lot longer, for the biggest subset more than 2 minutes pass before the end of the data loading. It's worth noting that the CPU usage is much lower than in the other dataset, always staying below 10%. These two things (time and CPU percentage) suggest us that the spark method to load the csv files is well parallelized, using many computing resources (CPU) but for a more limited period of time; on the other hand, the custom function used to load the json.gz file is not parallelized at all, parsing one line at a time, then consuming very less computing resources for a longer period of time.

4.3 Utility matrix preparation

The next step performed is the creation of the utility matrix which will be given as input to the ALS model. Figure 6 for MovieLens, Figure 7 for Amazon product data.

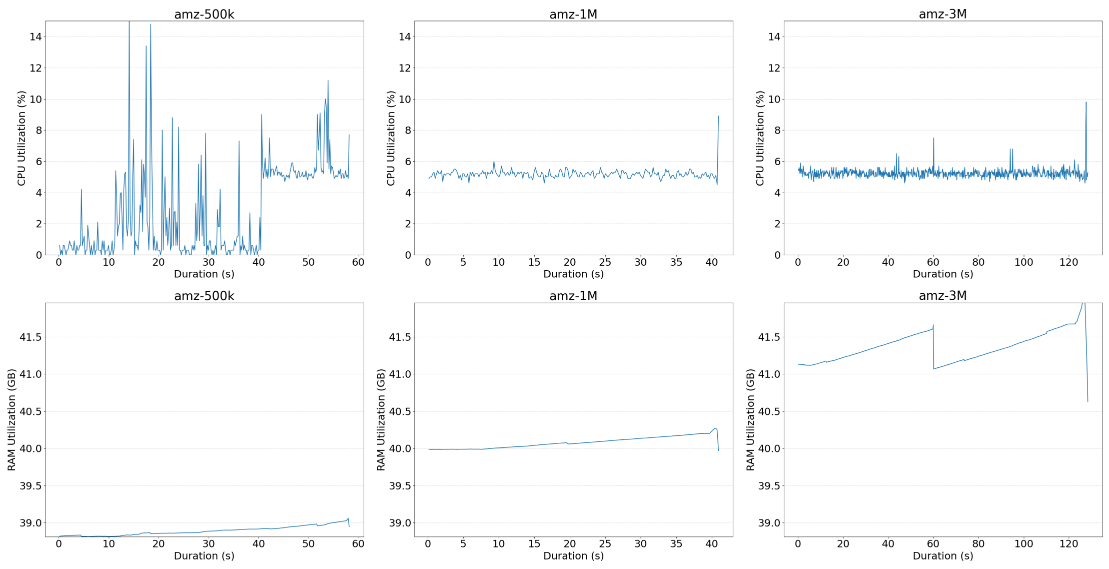

Figure 5: Statistics for second step (loading user ratings data) for Amazon product data dataset.

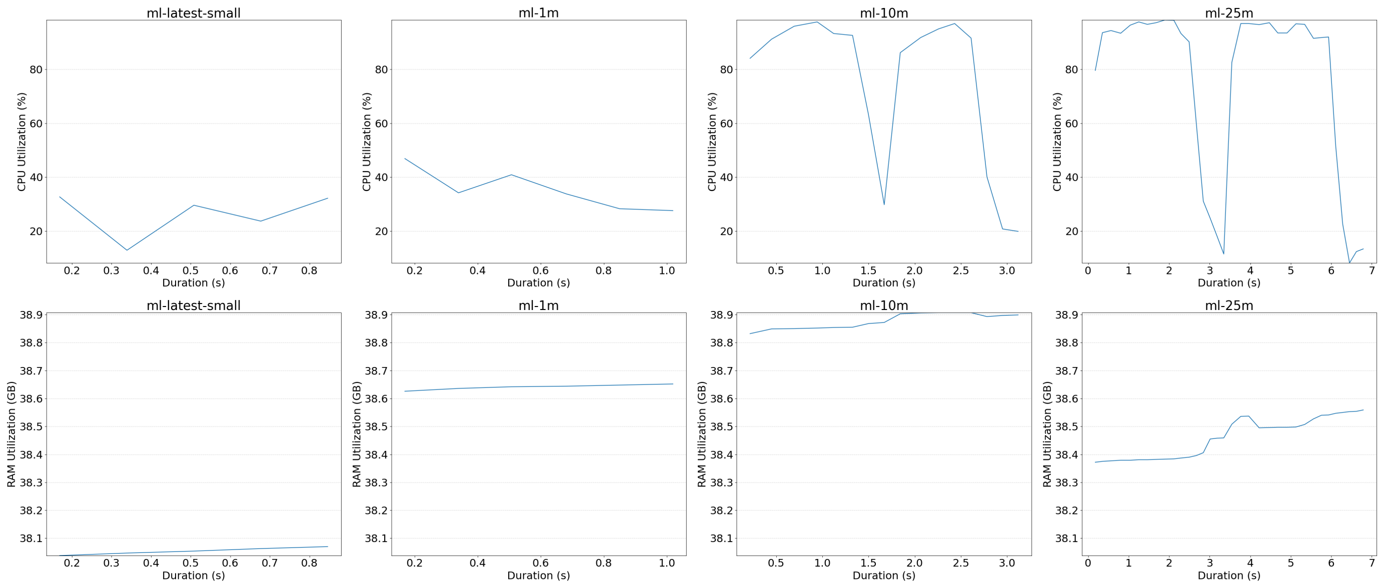

Figure 6: Statistics for third step (creating utility matrix for ALS) for MovieLens dataset.

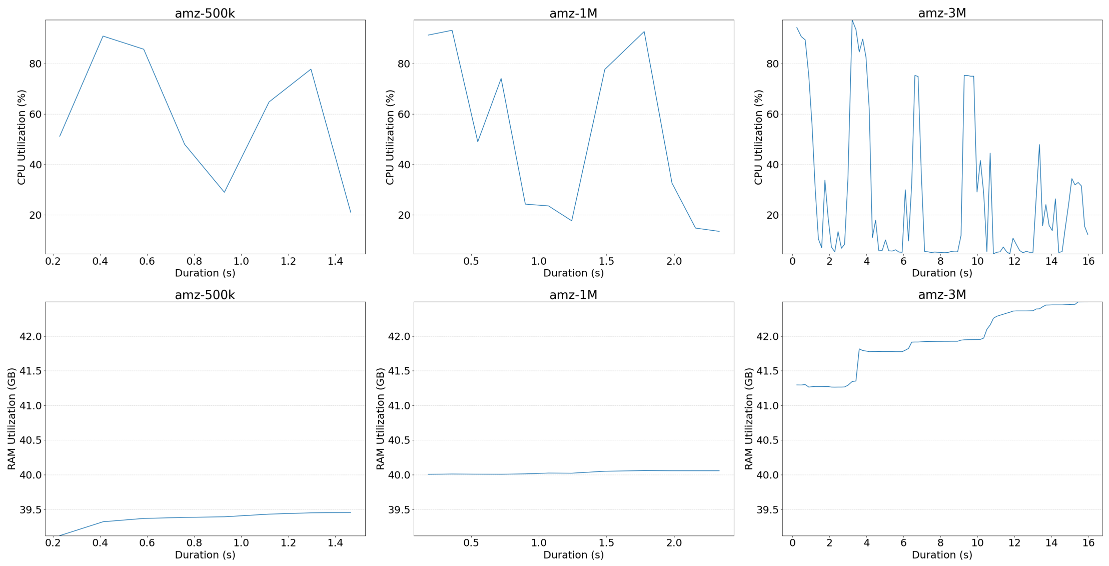

Figure 7: Statistics for third step (creating utility matrix for ALS) for Amazon product data dataset.

Watching the MovieLens statistics we can see one interesting thing: from the ml-1m subset to the ml-10m subset the increment of the number of ratings is 10× and for the number of movies is 2×; the time only triplicates, passing from 1 second to 3 seconds. This shows the efficacy of Spark parallelization. Similar thing between ml-10m and ml-25m, with “multiplicators” of 2.5× and 6× and elapsed time increased only by 2 times, from 3 to about 6 seconds.

We cannot observe the same for the Amazon product data dataset, where the previous reasoning is valid from amz-500k to amz-1M but not from amz-1M to amz-3M: for the least two subsets, in fact, for a number of user ratings of three times more we have an increment of duration of 8×. One possible explanation for this phenomenon could be the fact that the number of products (column of the matrix) is 500’000, about 8 times the number of movies in the largest MovieLens subset, and that could slow down a bit the computation.

4.4 ALS model training

For the ALS model training the results can be seen in Figure 8 for MovieLens and in Figure 9.

In the MovieLens statistics we can see a relevant increment of RAM usage only for the ml-25m subset (an increment of about 2 GB). The CPU usage is high (90%) at the start, especially for ml-10m and ml-25m, stabilizing around 50% after some seconds. What we can observe is that the time is not strictly linear w.r.t. the amount of data that the ALS have to process; that’s probably because of the performing parallelization of Spark.

For the Amazon product data results we can see that the subset that stresses the most both the CPU and RAM usage is the amz-3M set. also, we can see that, differently from the MovieLens dataset, the time grows almost exponentially

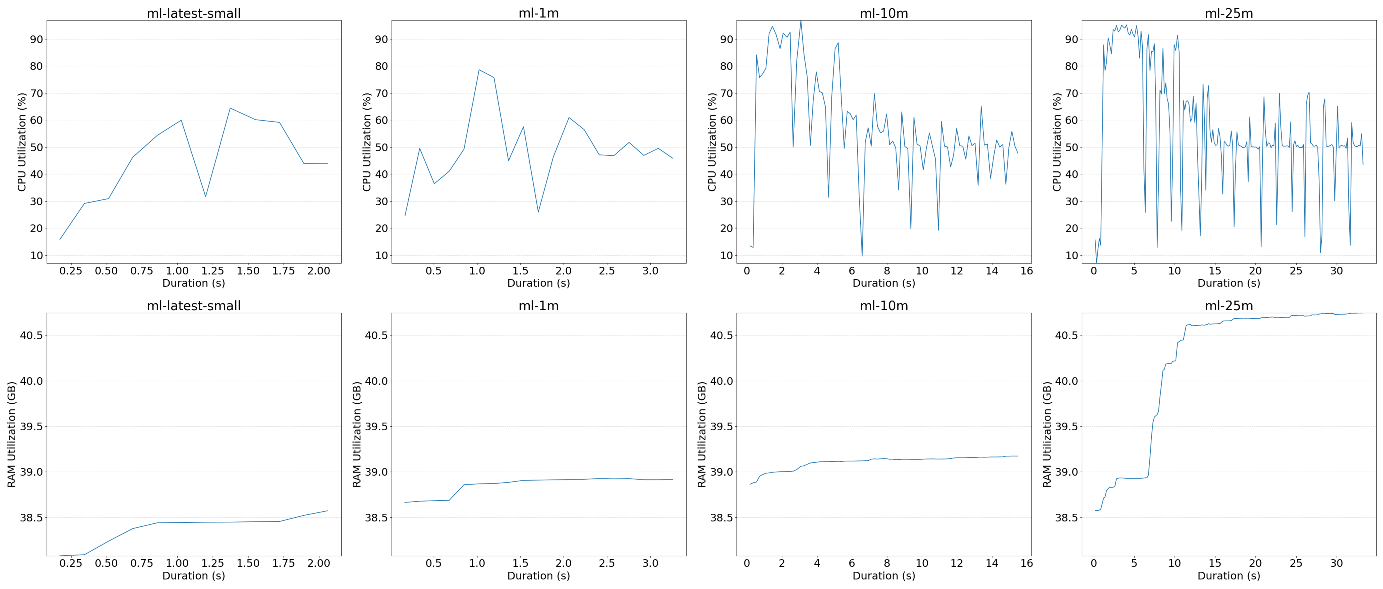

Figure 8: Statistics for fourth step (training the ALS model) for MovieLens dataset.

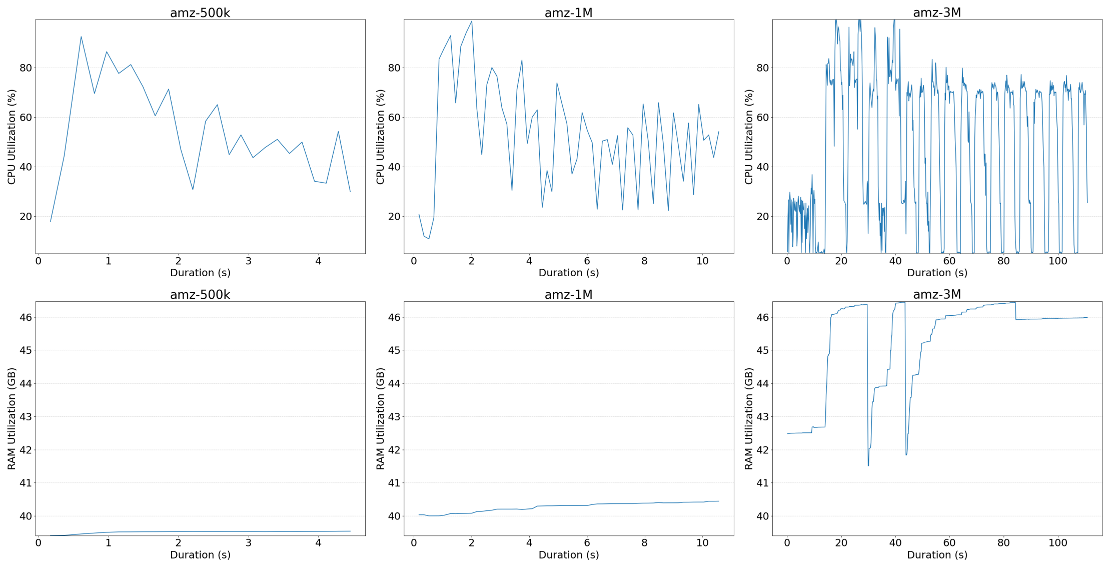

Figure 9: Statistics for fourth step (training the ALS model) for Amazon product data dataset.

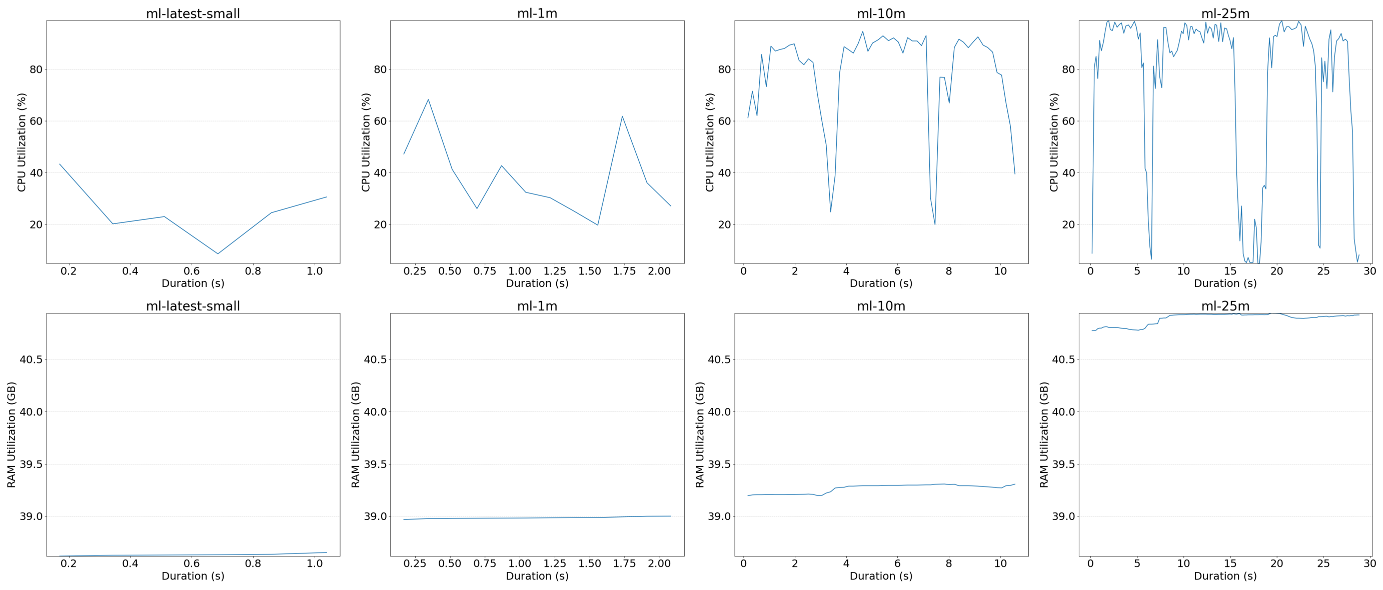

Figure 10: Statistics for fourth step (evaluation of the ALS model) for MovieLens dataset.

w.r.t. the amount of data. This difference can be explained with the fact that the ALS algorithm is strictly dependent on the type of utility matrix we are using, and not only in its dimensions. The variable that can influence its time of execution is, for example, the sparsity of the utility matrix.

4.5 ALS model evaluation

After the training we discuss about the ALS model evaluation: the statistics can be seen in Figure 10 for MovieLens and in Figure 11 for Amazon product data.

For MovieLens we can make almost the same considerations done for the training looking at elapsed time, CPU and RAM usage metrics. The only detail that changes is that in the ml-10m the CPU usage is more than 90% for a longer time than for the training phase, where the usage was intensive only in the first few seconds.

For Amazon data product, the observations are almost identical to the ones done for the training too. The duration is less but the proportions between data subsets is almost the same.

4.6 Read and write shuffle analysis

Results relative to the read and write shuffle and the number of tasks performed by Spark are shown in the Table 1. As said in Section 3, the following metrics has been retrieved w.r.t. all the five operations described before (aggregated), but distinctly for each data subset.

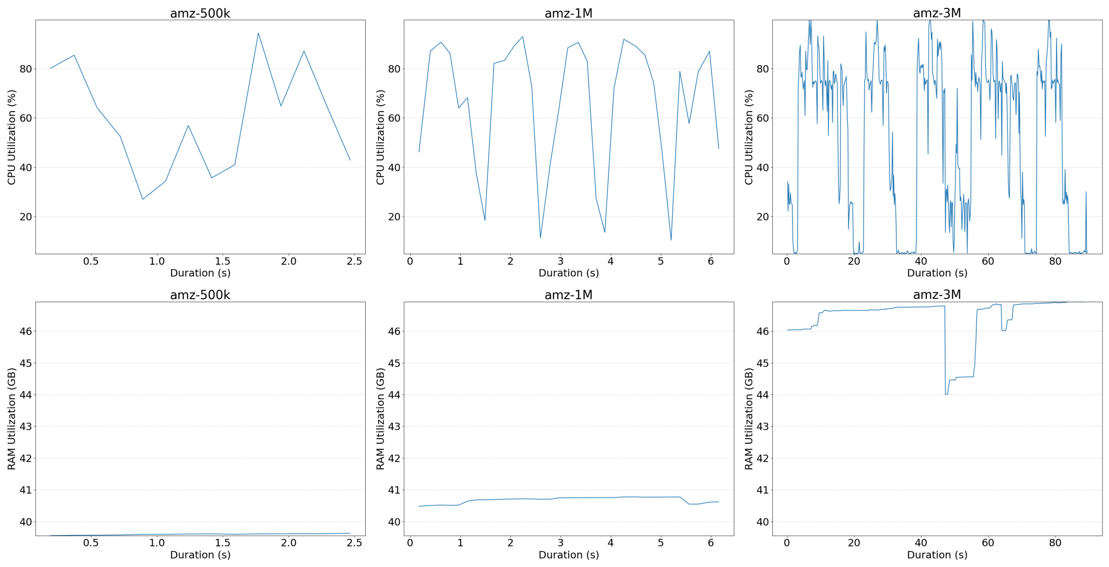

Figure 11: Statistics for fourth step (evaluation of the ALS model) for Amazon product data dataset.

We can notice that the total tasks in which the job are divided during their execution less than linearly w.r.t. the dataset dimension.

A different observation has to be done for what concerns the memory shuffle read and write: from the smallest to the biggest data subsets the used memory grow a lot more than linearly. That could be explained with the fact that, in order to handle a larger quantity of data, Spark has to exchange a lot more data in the network between stages.

5 Conclusions

Despite this project is about a recommendation system I focused more on Spark jobs performance, since it's a Big Data project. Analyzing the results helped me to better understand how changing the dataset dimensions and "format"

data subset	**total tasks**	**shuffle read**	**shuffle write**
ml-latest-small	224	26.1 MiB	26.8 MiB
ml-1m	276	74 MiB	81.5 MiB
ml-10m	430	691.4 MiB	748.9 MiB
ml-25m	474	3.4 GiB	4.9 GiB
amz-500k	506	223.7	311.2 MiB
amz-1M	566	715.2	887 MiB
amz-3M	576	3.1 GiB	3.4 GiB

Table 1: Report of results relative to Spark read and write shuffle and total Spark tasks.

can affect the execution of big data processing parallelization tools.

It's been challenging to adapt the recommendation system done for a products dataset to one that contains movies, and it was interesting to see the differences between the two and different subset of them, having to change some parameters in order to let Spark run correctly.

I learned how data can be integrated between tools, like Spark and elasticsearch and, most importantly, which are the most important aspects to consider when handling with such large datasets.