

# An Experimental Analysis on Scalable Implementations of the Alternating Least Squares Algorithm

Dânia Meira

Data Science Retreat/EyeEm  
Berlin, Germany

Email: meira.dania@gmail.com

José Viterbo

Institute of Computing  
Fluminense Federal University

Niterói, RJ, Brazil  
Email: viterbo@ic.uff.br

Flavia Bernardini

Institute of Computing  
Fluminense Federal University

Niterói, RJ, Brazil  
Email: fbernardini@ic.uff.br

**Abstract**—The use of the latent factor models technique overcomes two major problems of most collaborative filtering approaches: scalability and sparseness of the user’s profile matrix. The most successful realizations of latent factor models are based on matrix factorization. Among the algorithms for matrix factorization, alternating least squares (ALS) stands out due to its easily parallelizable computations. In this work we propose a methodology for comparing the performance of two parallel implementations of the ALS algorithm, one executed with MapReduce in Apache Hadoop framework and another executed in Apache Spark framework. We performed experiments to evaluate the accuracy of generated recommendations and the execution time of both algorithms, using publicly available datasets with different sizes and from different recommendation domains. Experimental results show that running the recommendation algorithm on Spark framework is in fact more efficient, once it provides in-memory processing, in contrast to Hadoop’s two-stage disk-based MapReduce paradigm.

## I. INTRODUCTION

AMONG the many techniques used to implement recommender systems, collaborative filtering, which is based on comparing the profile of preferences of the users, is a very popular technique in e-commerce applications, due to its good results [1]. Neighborhood-based approaches present scalability problems, given that the algorithm has to process all the data to compute a single prediction. Hence, if there is a large number of users and items, such approaches may not be appropriate for online systems which recommend in real time. Furthermore, these algorithms are more sensitive than model-based to some common problems of recommender systems. One common problem is the sparsity of the matrix that stores the ratings that represent the users’ preferences about the available items. This refers to a situation in which transactional or feedback data is sparse and insufficient to identify similarities in users’ interests making it difficult and unreliable to predict which consumers are similar [2]. Another recurrent problem in generating recommendations happens when we wish to recommend items that no one in the community has yet rated or interacted with. This is known as the cold-start problem and pure collaborative filtering cannot help in a cold start setting,

since no user preference information is available to form any basis for recommendations [3].

Nevertheless, there are models that can help bridge the gap from existing items to new items, by inferring similarities among them. Model-based approaches, instead of directly using the ratings stored, as the neighborhood-based systems, use ratings to learn a predictive model. The model building process is performed by different machine learning algorithms such as Bayesian networks [4], neural networks [5], and Singular Value Decomposition [6]. These approaches tend to be faster in prediction time than the neighborhood-based approaches. However, the construction of the model is a complex task that demands the estimation of a multitude of parameters, and usually requires a considerable amount of time [1].

These problems become more evident when trying to construct recommender systems associated with Websites that have a large number of users and items and, thus, associated with huge databases. Online systems demand high availability and short response time, as they must integrate and quickly process incoming streams of data from all users’ activities, in order to generate the recommendations. All this process need to occur with a latency of seconds, as the most promising items selected by the recommendation algorithms have to be showed to the users while they are still browsing the Website. The greater the number of users to serve and items from which to recommend, the greater is the amount of processing required, which increases the time it takes to generate each recommendation. The digital music Spotify platform [7] is a practical example of an online recommender system with high demand: their music personalization service has more than 50 million active users, 30 million cataloged songs and around 20 thousand new songs added per day [8]. Amazon [9] generates recommendations from a database with 253 million products [10] for users of 270 million active customer accounts [11]. An efficient approach is essential in all those cases. Nowadays, to tackle such performance challenges, online recommender systems have combined two strategies: (i) efficient algorithms, that avoid the computational complexity of calculating each of the entries of the high

dimensional and sparse matrix; and (ii) optimized data storage and processing. This means processing real-time information to build a predictive model and present its output in seconds.

In order to solve this problem, some authors have developed a class of model-based collaborative filtering algorithms that are fast and easy to calculate, called latent factor models [12]. They attempt to identify relevant features (latent factors) that explain observed ratings. These features can be interpreted as the preference of the users and the characteristics of the items being recommended. Using these latent factors, it is possible to infer the user's preference and make a recommendation of the better items for him/her. The most successful techniques to perform latent factors modeling are based on matrix factorization [13]. They have become popular recently because they combine scalability and predictive accuracy, and, besides, they offer flexibility for modeling different real situations, being superior to the neighborhood-based methods for producing recommendations because they allow the incorporation of additional information such as implicit feedback, temporal effects, and confidence levels [14]. Recent works suggest modeling only the observed ratings, while avoiding overfitting, through an adequate regularized model [15].

Some parallel algorithms for latent factor models with regularization have been designed aiming at improving the modelling performance. Among them, two can be highlighted: (i) the low-rank matrix factorization with Alternating Least Squares (ALS), which uses a series of broadcast-joins [16], built on top of the open source MapReduce implementation Hadoop [17], and its ecosystem, which we call HadoopMR-Mahout; and (ii) the Alternating Least Squares with Weighted- $\lambda$ -Regularization (ALS-WR) [18] which has been implemented in Apache Spark's Machine Learning library, MLlib, which we call Spark-MLlib [19]. Scalability and performance are key issues for recommender systems, since computational complexity increases with the number of users and items, but the performance gain for these implementations has not yet been systematically evaluated in any comparative study.

Although ALS Matrix Factorization algorithms are not new, some recent works shows that evaluating solutions that can be faster in specific situations, such as memory restrictions and some other high processing situations that may occur, still needs some attention. Authors of [20] propose some techniques for finding efficient and portable ALS Matrix Factorization for Recommender Systems. They apply thread batching technique and three architecture-specific optimizations for a new solution, and they implement an ALS solver in OpenCL so that it can run on various platforms (CPUs, GPUs, and MICs). Authors of [21] propose a new software solution to improve the performance of Recommender Systems, relying heavily on Apache Spark technology to speed up the computation of recommendation algorithms.

This work aims to conduct an experimental analysis to compare two different scalable implementations of the Alternating Least Squares algorithms (Spark-MLlib and HadoopMR-Mahout) for collaborative filtering recommendation. We performed experiments to evaluate the accuracy of generated

recommendations and the execution time of both algorithms, using publicly available datasets with different sizes and from different recommendation domains.

This work is organized as follows. In the next section, we explain the fundamental concepts about model-based approaches for implementing collaborative filtering. In Section 3, we discuss matrix factorization implementations, explaining how parallel implementations improve efficiency of recommender algorithms. In Section 4, we describe the methodology used for the comparative study between the two different implementations of the ALS algorithm, on different recommendation domains and dataset sizes, and present our experimental results for three assessed dimensions: accuracy, efficiency and scalability. In Section 5, we discuss the contributions and limitations of the proposed study, presenting also some topics for future work.

## II. MODEL-BASED COLLABORATIVE FILTERING

The fundamental assumption of CF is that if users  $X$  and  $Y$  rate  $n$  items similarly, or have similar behaviors (e.g., buying, watching, listening), hence they will rate or act on other items similarly. CF techniques use a database of preferences for items by users to predict additional topics or products a new user might like. The problem space can be formulated as a matrix of users versus items, with each cell representing a user's rating on a specific item. This matrix will be referred as ratings matrix from now on.

Under this formulation, the problem is to predict the values for specific empty cells. In collaborative filtering, this matrix is usually very sparse, since each user only rates a small percentage of the total available items. To fill in the missing entries of the ratings matrix, models are learnt by fitting the previously observed ratings. Once the goal is to generalize these observed ratings in a way that allows us to predict future, unknown ratings, caution should be exercised to avoid overfitting the observed data. This can be achieved by modeling the latent factors of the ratings matrix, that is, finding a small set of latent features that explain observed ratings and describe the general characteristics of users and items. The most successful techniques to model latent factors are based on matrix factorization, because they combine scalability and predictive accuracy.

### A. Matrix Factorization

Matrix factorization models map both users and items to a joint latent factor space, such that user-item interactions are modeled as inner products in that space. The latent space tries to explain ratings by characterizing both items and users on the same set of factors, which are the characteristics inferred from the observed ratings [18]. The intuition of this method is that it can be equivalent to a summarization. It boils down the world of user preferences for individual items to a world of user preferences for more general and less numerous features (like genre). This is, potentially, a much smaller set of data.

Although this process loses some information, it can sometimes improve recommendation results because this process

smooths the input in useful ways when it generalizes the features that describe the items, making similar what appeared to be distinct at first, thus avoiding overfitting the observed ratings. For example, imagine two car enthusiasts. One loves Corvette, and the other loves Camaro, and they want car recommendations. These enthusiasts have similar tastes: both love a Chevrolet sports car. But in a typical data model for this problem, these two cars would be different items. Without any overlap in their preferences, these two users would be deemed unrelated. However, a matrix factorization based recommender would perhaps find the similarity. The matrix factorization output may contain features that correspond to concepts like Chevrolet or sports car, with which both users would be associated. From the overlap in features, a similarity could be computed. These features correspond to the latent factors, or singular values of the ratings matrix and their correspondence to concepts are not explicit. Also, the exact number of singular values describing a matrix is not previously known, so there is a need to experiment to find the appropriate number of singular values that best summarizes the concepts for a given domain.

Consider a recommender system with  $m$  users and  $n$  items. Let  $R = [r_{ui}]$  be the ratings matrix, where  $r_{ui} \in \mathbb{R}^{(m \times n)}$ . Matrix factorization models map both users and items to a joint latent factor space of dimensionality  $k$ , that is,  $\hat{R}$  is a rank- $k$  approximation of the ratings matrix  $R$ . Let  $P = [p_u]$  be the user feature matrix, where  $p_u \in \mathbb{R}^k$ , and  $Q = [q_i]$  be the item feature matrix, where  $q_i \in \mathbb{R}^k$ . So, user-item interactions are modeled as inner products:

$$\hat{r}_{ui} = q_i^T \times p_u \quad (1)$$

An example of matrix factorization computation is found next on Figure 1. On a system with five users (represented by the upper matrix in the figure) and six items (represented by the left matrix in the figure), the estimation of the rating value of item 3 given by user 4,  $\hat{r}_{43}$ , is given by the inner product of the vectors representing item 3, i.e.,  $q_3^T$ , and user 4, i.e.,  $p_4$ .

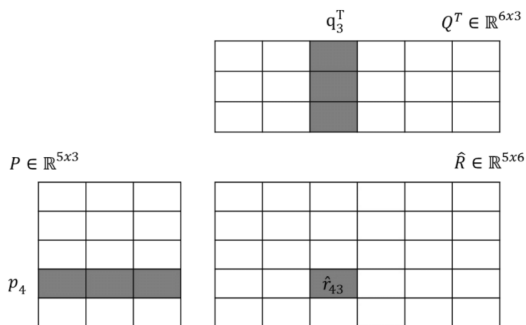


Fig. 1. Sinusoid

The major challenge is to compute the mapping of each item and user to latent factor vectors  $q_i, p_u \in \mathbb{R}^k$ . The traditional implementation to learn latent factors is the singular value decomposition (SVD), but it suffers from the high portion

of missing values in the ratings matrix, because, in general, users have rated only a small set of items [14]. The SVD can be computed one column at a time, whereas for the partially specified case, no such recursive formulation holds [18]. Also, addressing only few known ratings is highly likely to model overfitting [12]. Earlier works relied on imputation [2], [22], which fill in missing ratings and make the ratings matrix dense. However, the data may be considerably distorted by inaccurate imputation and also computing the SVD becomes very expensive after imputation, as it significantly increases the size of the matrices.

More recent works suggested modeling directly only the observed ratings, while avoiding overfitting through an adequate regularized model [15]. This model minimizes the regularized squared error on the set of observed ratings, as shown in Eq. 2, where  $k$  is the set of  $(u, i)$  pairs for which  $r_{ui}$  is known (the training set). The system learns the model by fitting the previously observed ratings. However, solving Eq. 2 with many parameters, when  $k$  is relatively large, from a sparse dataset usually overfits the data. The overfitting is avoided by regularizing the learning parameters, whose magnitudes are penalized by the  $\lambda$  constant [23]. This is also known as the Tikhonov regularization [24]. Eq. 2 is solved using a learning algorithm such as the alternating least squares (ALS) [18], which is the focus of this work.

$$\min(p, q) \sum_{(u, i) \in k} (r_{ui} - q_i^T p_u)^2 + \lambda (\|q_i\|^2 + \|p_u\|^2) \quad (2)$$

### B. Alternating Least Squares (ALS)

Eq. 2 is not convex when both  $q_i$  and  $p_u$  are unknown. However, fixing one of them turns the optimization into a quadratic problem that can be solved. So, ALS technique rotates between fixing the  $q_i$ 's and  $p_u$ 's. When all  $p_u$ 's are fixed, the system recomputes the  $q_i$ 's by solving a least-squares problem, and vice versa. This ensures that each step decreases Eq. 2 until convergence.

Although it is computationally more expensive than Stochastic Gradient Descent (SGD), ALS implementation is favorable in at least two cases. The first is when dealing with densely filled matrices, as such in systems centered on implicit data. Because the training set cannot be considered sparse, looping over each single training case (as in the case of SGD) would not be practical. The second case is when the system can use parallelization. The algorithm computes each  $q_i$  independently of the other item factors and computes each  $p_u$  independently of the other user factors, which allows for massive parallelization of the implementation [18].

When re-computing the user feature matrix  $P$  for example,  $p_i$ , the  $i$ -th row of  $P$ , can be re-computed by solving a least squares problem only including  $r_i$ , the  $i$ -th row of  $R$ , which holds user  $i$ 's interactions, and all the columns  $q_j$  of  $Q$  that correspond to non-zero entries in  $q_i$ . This re-computation of  $p_i$  is independent from the re-computation of all other rows of  $P$  and therefore, the re-computation of  $P$  is easy to parallelize if efficient data access to the rows of  $R$  and the corresponding

columns from  $Q$  is effectively managed. The sequence of re-computing of  $P$  followed by re-computing  $Q$  is referred to as a single iteration in ALS. Algorithm 1 summarizes the steps of the ALS algorithm.

From a data processing perspective, computing ALS means that a parallel join occurs between the interaction data  $R$  and  $Q$  (the item features) in order to re-compute the rows of  $P$ . Analogously, a parallel join is conducted between  $R$  and  $P$  (the user features) to re-compute  $Q$ . Finding an efficient execution strategy for these joins is crucial to the performance of any parallel solution, since the required amount of inter-machine communication, as network bandwidth is the scarcest resource in a cluster.

### III. MATRIX FACTORIZATION IMPLEMENTATIONS

In this section, we discuss previous works that propose matrix factorization implementations to solve the recommendation problem. We describe each of the proposed implementations and the performance results obtained.

#### A. Traditional Implementations

The Netflix Prize, a competition that began in October 2006, has motivated the progress in the field of collaborative filtering. The nature of the competition has encouraged rapid development, where innovators built on each generation of techniques to improve prediction accuracy. In September 2009 the prize was awarded to the BellKor's Pragmatic Chaos team that managed to achieve the winning RMSE of 0.8567 on the test subset, which represents a 10.06% improvement over Cinematch, Netflix's own recommendation algorithm.

The recommendation strategy used by the winning solution was an ensemble of more than 100 different predictor sets, the majority of which are factorization models, learned by stochastic gradient descent (SGD), applied directly on the raw data.

For single machine implementations, SGD is the preferred technique to compute a low-rank matrix factorization, because it is easy to implement and computationally less expensive than ALS. Unfortunately, SGD is inherently sequential, because it updates the model parameters after each processed interaction. Techniques for parallel SGD have been proposed, yet they are either hard to implement, exhibit slow convergence or require shared-memory.

The SGD implementation used in this solution is described by [25] as possible to be executed to factorize the 17,000 x 500,000 matrix with 40 latent factors on 2G of RAM, a C compiler, and good programming habits. But in the paper describing the winning solution, he did not specify the environment nor the performance of the algorithm, as this was not important for the prize. The algorithms could run for as many as long as needed, since the only evaluated metric was the RMSE.

Finally, in 2012 Netflix announced that they did not implement the Netflix Prize solution algorithm, and they gave two reasons for that. The first reason is that the new methods were evaluated off-line but the additional accuracy gains measured

did not seem to justify the engineering effort needed to bring them to a production environment. Also, their focus on improving personalization had shifted since 2007, just a year after the beginning of the competition, when Netflix streaming service was launched. From DVDs to an online streaming service, Netflix as a whole changed dramatically, not only the way the users interact with the service but also the types of data available to use in the algorithms.

As of 2012, Netflix reported having more than 23 million subscribers in 47 countries. Those subscribers streamed 2 billion hours from hundreds of different devices in the last quarter of 2011. Every day they add 2 million movies and TV shows to the queue and generate 4 million ratings. They have adapted their recommendation algorithm to this new scenario, and 75% of what people watch is from some sort of recommendation. This new strategy still runs the learning algorithm in batch, as briefly discussed in the Large-Scale Recommendation Systems Workshop on the ACM Conference Series on Recommender Systems in 2013, held at Hong Kong.

#### B. Parallel Implementations

Another team participating in the Netflix Prize proposed, in 2008, a parallel implementation of matrix factorization, called the Alternating-Least-Squares with Weighted- $\lambda$ -Regularization (ALS-WR) [18]. This solution was motivated by two main reasons: the size of the dataset, which was 100 times larger than previous benchmark datasets, resulting in much longer model training time and much larger system requirements; and the fact that the observed ratings corresponded to only about 1% of the complete ratings matrix, which means dealing with a very sparse matrix. Since this implementation was motivated by the Netflix data, it is dealing with observed ratings, or explicit feedback. Thus, it solves the matrix factorization problem with ALS using only the observed ratings. Rewriting Eq. 2, Eq. 3 is obtained, where  $n_{m_i}$  and  $n_{m_u}$  are the number of observed ratings for the item  $i$ , and for the user  $u$  respectively. Let  $I_u$  denote the set of items  $i$  that user  $u$  has rated, then  $n_{m_u}$  is the cardinality of  $I_u$ ; similarly  $I_i$  denotes the set of users who rated item  $i$ , and  $n_{m_i}$  is the cardinality of  $I_i$ .

$$\begin{aligned} \min(p, q) \sum_{(u,i) \in k} (r_{ui} - q_i^T p_u)^2 \\ + \lambda \left( \sum_i n_{m_i} \|q_i\|^2 + \sum_u n_{m_u} \|p_u\|^2 \right) \end{aligned} \quad (3)$$

The solution for Eq. 3 follows the steps demonstrated in Section II-B, but, instead of initializing the matrix  $Q$  to random values on Step 1 in Alg. 1, it suggests assigning the average rating for that item as the first row, and small random numbers for the remaining entries. The stopping criterion used is based on the observed RMSE on the validation dataset. After one round of updating both  $Q$  and  $P$ , if the difference between the observed RMSEs is less than 0.0001, the iteration stops and the obtained  $P$ ,  $Q$  are used to make final predictions on the test dataset.

**Algorithm 1** ALS algorithm

---

```

1: procedure ALS( $P, Q$ ) ▷ Matrices representing user feature matrix and item feature matrix, respectively
2:   Initialize matrix  $Q$  with random values
3:   repeat
4:     Fix  $Q$ , solve  $P$  by minimizing the objective function (the sum of squared errors)
5:     Fix  $P$ , solve  $Q$  by minimizing the objective function similarly
6:   until Stop criteria is satisfied
7:   return  $P, Q$ 
8: end procedure

```

---

In this cited approach, a version that allows for parallel computation of Matlab was used. It creates several separate copies of Matlab, each with its own private workspace, and each running on its own hardware platform, collaborate and communicate to solve problems. Each such running copy of Matlab is referred to as a “lab”, with its own identifier (labindex) and with a static variable (numlabs) telling how many labs there are. Matrices can be private (each lab has its own copy, and their values differ), replicated (private, but with the same value on all labs) or distributed (there is one matrix, but with rows, or columns, partitioned among the labs).

Because all of the steps use  $R$ , two distributed copies of it were used: one distributed by rows (i.e., by users) and the other by columns (i.e., by items). Both  $P$  and  $Q$  matrices were distributed computed and updated. While computing  $P$ , it is required a replicated version of  $Q$ , and vice versa. Thus, the labs communicate to make the replicated versions of these matrices from the distributed versions that are first computed. Matlab’s “gather” function performs the inter-lab communication needed for this.

To update  $Q$ , it is required a replicated copy of  $P$ , local to each lab. The ratings data distributed by columns (items) is used. The data is distributed by blocks of equal numbers of items. The lab that stores the ratings of item  $i$  will, naturally, be the one that updates the corresponding column of  $Q$ , which is items  $i$ ’s feature vector. Each lab computes  $q_i$  for all items in the corresponding item group, in parallel. These values are then “gathered” so that every node has all of  $Q$ , in a replicated array. To update  $P$ , similarly all users are partitioned into equal-size user groups and each lab just updates user vectors in the corresponding user group, using the ratings data partitioned by rows.

The broadcast step is the only communication cost due to using a distributed, as opposed to a shared-memory, algorithm. This method reported taking up less than 5% of the total run time. The algorithm achieves a nearly linear speedup; for  $k = 100$ , it takes 2.5 hours to update  $P$  and  $Q$  once with a single processor, as opposed to 5 minutes with 30 processors.

This first work implemented ALS in parallel Matlab and executed on a Linux cluster, with 30 Xeon 2.8GHz processors and every four processors shared 6 GB of RAM. When applied to the Netflix dataset with 100 latent factors and 30 iterations was computed in 2.5 hours and obtained a RMSE of 0.8985 which is a performance improvement of 5.91% over Netflix’s Cinematch system.

After the popularization of the Hadoop platform, the parallelization of the ALS algorithm was revisited with a new proposal for a parallel implementation using a series of broadcast-joins that can be efficiently executed with MapReduce [16]. This implementation has partially contributed to Apache Mahout, the open source machine learning library that runs on top of Apache Hadoop framework, and is publicly available. The evaluation setup was a cluster of 26 machines, each with two 8-core Opteron CPU and 32GB of RAM. The experiments showed that on the Netflix dataset, which consists of more than a million ratings given to 17,700 movies by 480,189 users, it was possible to run 37 to 47 iterations of the algorithm, and it typically converges after 15 iterations [18].

This approach is limited to use-cases where neither  $Q$  nor  $P$  need to be partitioned, meaning they individually fit into the memory of a single machine of the cluster. A rough estimate of the required memory for the re-computation steps in ALS is  $\max(|M|, |N|) \times k \times 8\text{byte}$ , as alternatively, a single dense double precision representation of the matrices  $Q$  or  $P$  has to be stored in memory on each machine. Even for 10 million users or items and a rank  $k = 100$ , the estimated required memory would be less than 8 GB, which can easily be handled by today’s commodity hardware. Experiment results show that, despite this limitation, this implementation is able to handle datasets with billions of data points.

In such a setting, an efficient way to implement the necessary joins for ALS in MapReduce is to use a parallel broadcast-join. The smaller dataset ( $Q$  or  $P$ ) is replicated to every machine of the cluster. Because all of the steps use  $R$ , each machine already holds a local partition of  $R$  which is stored in the DFS. Then the join between the local partition of  $R$  and the replicated copy of  $P$  (and analogously between the local partition of  $R$  and  $Q$ ) can be executed by a map operator. This operator can additionally implement the logic to re-compute the feature vectors from the join result, which means that it is possible to execute a whole re-computation of  $Q$  or  $P$  with a single map operator.

Figure 2 illustrates the parallel join for re-computing  $P$  using three machines. First, the broadcast of  $Q$  is done to all participating machines, which create a hashtable for its contents, the item feature vectors.  $R$  is stored in the DFS partitioned by its rows and forms the input for the map operator, where e.g.,  $R(1)$  refers to partition 1 of  $R$ . The map operator reads a row  $r_i$  of  $R$  (the interaction history of user  $i$ ) and selects all the item feature vectors  $q_j$  from the

hashtable holding  $Q$  that correspond to non-zero entries  $j$  in  $r_i$ . Next, the map operator solves a linear system created from the interactions and item feature vectors and writes back its result, the re-computed feature vector  $p_i$  for user  $i$ . The re-computation of  $Q$  works analogously, with the only difference that  $P$  is broadcasted and  $R$  is stored with partitioning done by its columns (the interactions per item) in the DFS.

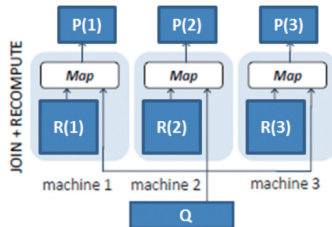


Fig. 2. Parallel recomputation of user features by a broadcast join [16]

This proposed approach is able to avoid some of the drawbacks of MapReduce and the Hadoop implementation described in Section III-A. It uses only map jobs that are easier to schedule than jobs containing map and reduce operators. Additionally, the costly shuffle-phase is avoided, in which all data would be sorted and sent over the network, once the join and the re-computation are done in a single job, which also spare to materialize the join result in the HDFS. This implementation contains multithreaded mappers that leverage all cores of the worker machines for the re-computation of the feature matrices and uses JBlas for solving the dense linear systems present in ALS. The broadcast of the feature matrix is conducted via Hadoop’s distributed cache in the initialization phase of each re-computation. Furthermore, Hadoop is configured to reuse the VMs on the worker machines and cache the feature matrices in memory to avoid that later scheduled mappers have to reread the data. The main drawback of a broadcast approach is that every additional machine in the cluster requires another copy of the feature matrix to be sent over the network.

The implementation was also validated on a synthetic dataset called Bigflix, generated from the Netflix dataset and containing 25 million users and more than 5 billion ratings. The performed scale-out test measured the average runtime per job during 5 iterations with 10 latent factors on clusters of 5, 10, 15, 20 and 15 machines. With 5 machines iteration takes about 19 minutes and with 25 machines it was 6 minutes faster.

#### IV. EXPERIMENTAL ANALYSIS

In this section, we give details of the ALS implementations on Mahout and MLlib libraries that will be executed on Hadoop and Spark respectively. We describe the datasets and the ambient on which the implementations are evaluated, and present the experimental evaluation on the parallel implementations.

#### A. Datasets

The datasets chosen to run the experiments are from the movies domain (MovieLens) and jokes domain (Jester). Due to copyright problems, Netflix dataset is not available for download anymore. So, to perform the recommendation evaluation on the movies domain, the MovieLens data is frequently used. The MovieLens dataset consists of anonymous ratings of movies and contains approximately 10 million ratings from 71,567 users on 10,681 movies. Ratings are made on a 5-star scale (whole-star ratings only) and each user has at least 20 ratings. The dataset was collected and made available by GroupLens Research, which currently operates a movie recommender based on collaborative filtering, at their webpage. This dataset was previously used to evaluate matrix factorization based methods with neighbor based correction technique, and achieved a best RMSE score of 0.8275 [13].

The Jester dataset consists of anonymous ratings of jokes collected between November 2006 and May 2009. Thus, this data is in a humor domain. It was firstly used to test the Eigentaste recommender and now is freely available for research use. The full data set contains 1,761,439 ratings from 59,132 users on 140 jokes. The ratings are real values ranging from -10.00 to +10.00. Ten percent of the jokes (called the gauge set, which users were asked to rate) are densely rated, others, more sparsely. Two thirds of the users have rated at least 36 jokes. The remaining users have rated between 15 and 35 jokes. The average number of ratings per user is 46, so it is a particularly dense data set compared to Netflix Prize and MovieLens. This dataset was previously used to evaluate matrix factorization based methods with neighbor based correction technique, and achieved a best RMSE score of 4.1229 [13].

#### B. Mahout ALS Implementation

Mahout 0.9, which was used for this evaluation, presents a MapReduce implementation of ALS that is composed of two jobs: a parallel matrix factorization job, which contains training phase of the ALS algorithm, and a recommendation job that outputs a list of recommended item ids for each user.

Given the ratings matrix ( $R$ ), the matrix factorization job computes the two intermediate matrices: user-to-feature ( $P$ ) and item-to-feature ( $Q$ ). This implementation follows the strategy described in Section III-B, the parallel broadcast-join [16]. Firstly, the smaller dataset ( $Q$  or  $P$ ) is replicated to every machine of the cluster. Also, the ratings matrix is partitioned, and each partition sent to a machine on the cluster, which stores it in the local HDFS. The join between the local partition of  $R$  and the replicated copy of  $P$  (and analogously between the local partition of  $R$  and  $Q$ ) can be executed by a map operator. This operator can additionally implement the logic to re-compute the feature vectors from the join result, which means that it is possible to execute a whole re-computation of  $Q$  or  $P$  with a single map operator.

The recommendation job processes the user-to-feature matrix and item-to-feature matrix calculated from the factorization job to compute the top-N recommendations per user. The

```

val Rb = spark.broadcast(R)
for (i <-1 to ITERATIONS){
  P = spark.paralellize(0 until n)
    .map(j => updateUser(j, Rb, Q))
    .collect()
  Q = spark.paralellize(0 until m)
    .map(j => updateUser(j, Rb, P))
    .collect()
}

```

Fig. 3. ALS Spark implementation

predicted rating between user and item is a dot product of the user's feature vector and the item's feature vector.

### C. MLib ALS Implementation

This is a blocked implementation of the ALS factorization algorithm that groups the two sets of factors (referred to as "users" and "items") into blocks and reduces communication by only sending one copy of each user vector to each item block on each iteration, and only for the item blocks that need that user's feature vector. This is achieved by precomputing some information about the ratings matrix to determine the "out-links" of each user (which blocks of items it will contribute to) and "in-link" information for each item (which of the feature vectors it receives from each user block it will depend on). This allows the implementation to send only an array of feature vectors between each user block and item block, and have the item block find the users' ratings and update the items based on these messages.

Because all of the steps use the ratings matrix  $R$ , it is helpful to make it a broadcast variable so that it does not get re-sent to each node on each step. Figure 3 shows the ALS Spark implementation. Note in Lines 3 to 5 that collection 0 until  $u$  are parallelized and collected to update each array [26]. The ALS recommender accepts as input an RDD (Resilient Distributed Datasets) of ratings (user: Int, product: Int, rating: Double).

### D. Experimental Results

The experiments were developed with Python 2.6 and firstly executed in local single machine mode for testing. Then, the final experiments were executed at Amazon Web Services (AWS).

The clusters used for the evaluation consists of t2.small EC2 instances running Ubuntu 64-bit OS with Oracle Java (JDK) 7, Apache Hadoop 1.2.1 and Apache Spark 1.1.1. Each t2.small instance has a 3.3GHz core processor, 2GB of RAM and 15GB of SSD storage. The accuracy and efficiency experiments were conducted on a cluster of 4 machines.

To evaluate these algorithms, the datasets were randomly divided into three non-overlapping subsets, named: training (60%), test (20%), and validation (20%). These datasets are saved on two datanodes of the HDFS, since this is the smaller cluster configuration for scalability experiment.

These two datanodes are accessible for all the workers through the experiments, since Spark is running in the same Hadoop cluster through Spark's standalone mode, that is, by simply placing a compiled version of Spark on each node on the cluster.

1) *Accuracy and Efficiency Experiment:* To evaluate the quality of the recommendations produced by each of the two implementations, multiple models are trained based on the training set, and that which achieves the smallest root-mean-square error (RMSE), given by Eq. 4, on the validation set after running 20 iterations of the algorithm is chosen as the best fit ALS model [18]. Finally, this model is evaluated on the test set.

$$RMSE = \sqrt{\frac{1}{|S_{val}|} \sum_{(m,n) \in S_{val}} (r_{ui} - \hat{r}_{ui})^2} \quad (4)$$

The parameters tested to find the best fit ALS model are combinations resulting from the cross product of the dimensionality of the latent factor space,  $k = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$  and the regularization parameter  $\lambda = [0.10, 0.25, 0.50, 0.75, 1.00]$ .

Analyzing the convergence of the Spark-MLlib ALS on the Jester validation set for the number of latent factors ( $k$ ), we can see that the recommendation quality usually improves when increasing  $\lambda$  until the optimal value of 0.5. Then, the increment worsens the recommendation accuracy. For the HadoopMR-Mahout ALS on the Jester validation set, we observed the same behavior presented by the Spark-MLlib implementation: the recommendation quality usually improves when increasing  $\lambda$  until the optimal value of 0.5 but beyond that, the recommendation is worse.

The best fit Spark-MLlib ALS for the Jester dataset has RMSE on the test set of 4.1339 and 4.1395 for the HadoopMR-Mahout. Both models have the same value for the parameters  $k$  and  $\lambda$ , but the MLib implementation achieves a result 0.13% better, with a training execution time that is more than 10 times faster, in a cluster with 4 t2.small instances, as shown on Table I.

TABLE I  
BEST FIT ALS MODEL RESULTS FOR JESTER DATASET

	Spark-MLlib	HadoopMR-Mahout
$k_{BestFitALS}$	20	20
$\lambda_{BestFitALS}$	0.5	0.5
RMSE(Validation Set)	4.1378	4.1385
RMSE(Test Set)	4.1339	4.1395
Execution time (sec)	61.4	671.4

For the MovieLens dataset, the best fit Spark-MLlib ALS is trained with  $k = 20$ ,  $\lambda = 0.5$  and RMSE = 0.8099 on the validation set. We observed that the model converges on each value of  $\lambda \geq 0.5$  regardless of the feature space size. The RMSE on the test set is 0.8091, which means that the model does not overfit the observed ratings. For the HadoopMR-Mahout ALS modelling results, the best fit is trained with



$k = 20$ ,  $\lambda = 0.5$ , and  $RMSE = 0.8196$  on the validation set, the same parameters found for the Spark-MLlib implementation. The convergence behavior found before repeats itself here, for each  $\lambda \geq 0.5$  regardless of the feature space size. Also, the RMSE on both implementations for the same regularization parameter is very close: the largest difference is of only 0.0002 or 0.001% of the rating score, represented on a scale of -10.0 to +10.0.

Comparing the best fit ALS models achieved by both implementations, again the Spark-MLlib solution has a better performance: more accurate, with a RMSE on the test set 1.4% smaller than the HadoopMR-Mahout implementation, and more efficient, with execution more than 5 times faster to run (Fig. 4). The results for the MovieLens dataset are summarized on Table II.

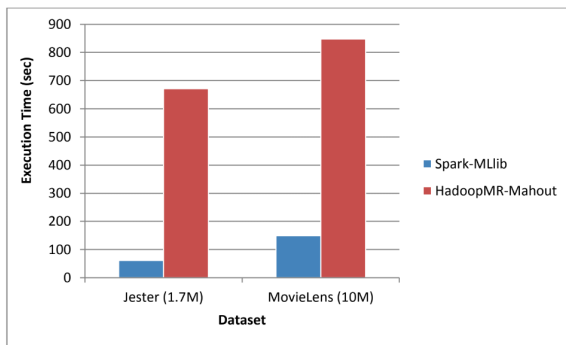


Fig. 4. Execution time for the best fit ALS model on a cluster with 4 machines

TABLE II  
BEST FIT ALS MODEL RESULTS FOR MOVIELENS DATASET

	Spark-MLlib	HadoopMR-Mahout
$k_{Best\ fit\ ALS}$	20	20
$\lambda_{Best\ fit\ ALS}$	0.1	0.1
RMSE(Validation Set)	0.8099	0.8196
RMSE(Test Set)	0.8091	0.8202
Execution time (sec)	149.1	847.9

2) *Scalability Experiment*: To test the scalability of these recommender systems, we measure the walltime of 20 iterations of the best fit ALS model on each of the datasets on different cluster sizes, consisting of 2, 4 and 6 AWS EC2 t2.small instances. We observe that the computation speedup does not linearly scale with the number of machines, which is an expected behavior since both implementations have a broadcast of the ratings matrix so every additional machine causes another copy of it to be sent over the network. Comparing the speedup values for the two implementations, shown on Fig. 5, we find that, when training the HadoopMR-Mahout ALS model with 6 machines, it shows an improvement of 1.60x on the Jester dataset and 1.45x on the MovieLens dataset over the execution with 2 machines, and for the Spark-MLlib implementation, executing with 6 machines provides an improvement of 1.86x on the Jester dataset and 2.39x on the MovieLens dataset over the execution with 2 machines.

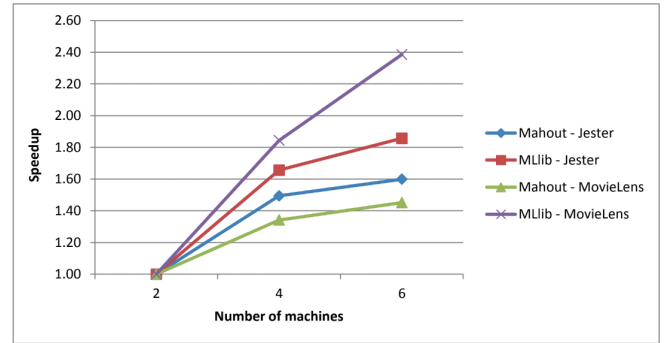


Fig. 5. Speedup for both implementations

As seen in Table III, the distributed and parallel ALS implementation on MLib executed on the Spark cluster with 6 machines achieved the faster training time for both datasets: 54.7 seconds for Jester that contains about 1.7 million joke ratings, and 115.3 seconds for MovieLens that contains about 10 million movie ratings. By extrapolating these results, we find that a recommender system with a dataset with 100 million ratings input, which is 10 times bigger than the MovieLens dataset, would take about 415 seconds to be trained on a cluster with 6 machines with the t2.small EC2 configuration. If we wished to put such a system into production, we could either utilize more of these general purpose instances or choose machines with more RAM, such as the M3 instances or the R3 memory optimized instances, which suggests that the Spark implementation is suitable for real world use cases.

TABLE III  
SUMMARY OF RECOMMENDATION TIME

Dataset	size (# of ratings)	Recommendation time (in sec)
Jester:	1.7 mi	54.7
MovieLens:	10 mi	115.3
	100 mi	415

## V. CONCLUSIONS

Alternating Least Squares (ALS) algorithm is an efficient approach in situations where generating online recommendations and processing large datasets is required. In this work, we described two scalable parallel implementations of the ALS algorithm, the Mahout ALS and MLib ALS. Each one uses a different framework for distributed processing on clusters of commodity hardware, respectively, Hadoop MapReduce and Spark.

We performed an experimental analysis comparing the different implementations of the ALS algorithm for collaborative filtering recommender systems, using datasets from two different domains: MovieLens, from the movies domain, and Jester, from the jokes domain. First we found the best fit ALS model for each of the datasets. Using the optimized parameters to train the ALS models, we performed the evaluation of the implementations in terms of execution time and accuracy results on the test set.



The experimental results showed that Spark-MLlib solution has a better performance than the Mahout ALS in terms of accuracy and efficiency for both recommendation domains. For the Jester dataset, the RMSE on the test set with Spark-MLlib was 0.13% better than with HadoopMR-Mahout, and the training was more than 10 times faster in a cluster with 4 machines. For the MovieLens dataset, the RMSE on the test set was 1.4% smaller on the Spark-MLlib implementation, and the modeling was 5 times faster. This study also featured a scalability experiment, running the best fit ALS model on clusters of 2, 4 and 6 machines. Again, the results were favorable to Spark, since it has a more expressive computational speedup: training time on a cluster with 6 machines was 86% faster on the Jester dataset and 139% on the MovieLens dataset when comparing to execution time on a cluster with 2 machines.

Deploying a recommender system on six t2.small instances available from EC2 took 115.3s for a dataset containing about 10 million ratings, and, by extrapolation, it would take about 415s for a dataset with 100 million ratings. The results suggest that a cluster with at least six t2.small instances or fewer and more potent machines, like M3 or R3 memory optimized instances available on EC2, would run a user's full recommendations measures in a few seconds, which is a suitable time frame for production settings. Future works are desirable in order to keep comparing the recommendation algorithms implementations available in the newer releases of MLlib and Mahout, as well as newer technologies, since both engines for large-scale data processing are rapidly evolving.

#### REFERENCES

- [1] F. Ccheda, V. Carneiro, D. Fernández, and V. Formoso, "Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems," *ACM Trans. Web*, vol. 5, no. 1, pp. 2:1–2:33, Feb. 2011. doi: 10.1145/1921591.1921593. [Online]. Available: <http://doi.acm.org/10.1145/1921591.1921593>
- [2] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th International Conference on World Wide Web*, ser. WWW '01. New York, NY, USA: ACM, 2001. doi: 10.1145/371920.372071. ISBN 1-58113-348-0 pp. 285–295. [Online]. Available: <http://doi.acm.org/10.1145/371920.372071>
- [3] Z. Huang, H. Chen, and D. Zeng, "Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering," *ACM Trans. Inf. Syst.*, vol. 22, no. 1, pp. 116–142, Jan. 2004. doi: 10.1145/963770.963775. [Online]. Available: <http://doi.acm.org/10.1145/963770.963775>
- [4] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN 1-55860-555-X pp. 43–52. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2074094.2074100>
- [5] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted boltzmann machines for collaborative filtering," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007. doi: 10.1145/1273496.1273596. ISBN 978-1-59593-793-3 pp. 791–798. [Online]. Available: <http://doi.acm.org/10.1145/1273496.1273596>
- [6] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. T. Riedl, "Application of dimensionality reduction in recommender system – a case study," in *WebKDD Workshop, held in conjunction with the ACM-SIGKDD Conference on Knowledge Discovery in Databases (KDD'2000)*, 2000.
- [7] Spotify, "Spotify," n.d., available at <https://www.spotify.com/>.
- [8] C. Johnson, "Scala data pipelines for music recommendations," 2015, available at <http://www.slideshare.net/MrChrisJohnson/scala-data-pipelines-for-music-recommendations>.
- [9] Amazon.com, "Amazon.com," n.d., available at <http://www.amazon.com/>.
- [10] ExportX, "How many (more) products does amazon sell?" 2014, available at <http://export-x.com/2014/08/14/many-products-amazon-sell-2>.
- [11] Statista, "Number of worldwide active amazon customer accounts from 1997 to 2014 (in millions)," 2014, available at <http://www.statista.com/statistics/237810/number-of-active-amazon-customer-accounts-worldwide/>.
- [12] Y. Koren and R. Bell, "Advances in collaborative filtering," in *Recommender Systems Handbook*, F. Ricci, L. Rokach, B. Shapira, and P. Kantor, Eds. Springer, 2011, pp. 33–48.
- [13] G. Takács, I. Pilászy, B. Németh, and D. Tikk, "Using visual representations of data to enhance sensemaking in data exploration tasks," *Journal of Machine Learning Research*, vol. 10, pp. 623–656, 2009.
- [14] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, Aug. 2009. doi: 10.1109/MC.2009.263. [Online]. Available: <http://dx.doi.org/10.1109/MC.2009.263>
- [15] R. Bell, Y. Koren, and C. Volinsky, "Modeling relationships at multiple scales to improve accuracy of large recommender systems," in *Proc. 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07. New York, NY, USA: ACM, 2007. doi: 10.1145/1281192.1281206. ISBN 978-1-59593-609-7 pp. 95–104. [Online]. Available: <http://doi.acm.org/10.1145/1281192.1281206>
- [16] S. Schelter, C. Boden, M. Schenck, A. Alexandrov, and V. Markl, "Distributed matrix factorization with mapreduce using a series of broadcast-joins," in *Proceedings of the 7th ACM Conference on Recommender Systems*, ser. RecSys '13. New York, NY, USA: ACM, 2013. doi: 10.1145/2507157.2507195. ISBN 978-1-4503-2409-0 pp. 281–284. [Online]. Available: <http://doi.acm.org/10.1145/2507157.2507195>
- [17] The Apache Software Foundation, "Apache hadoop," n.d., available at <http://hadoop.apache.org/>.
- [18] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Algorithmic Aspects in Information and Management. AAIM 2008. LNCS, vol 5034*, R. Fleischer and J. Xu, Eds. Springer, 2008.
- [19] The Apache Software Foundation, "MLlib," n.d., available at <http://spark.apache.org/mllib/>.
- [20] J. Chen, J. Fang, W. Liu, T. Tang, X. Chen, and C. Yang, "Efficient and portable als matrix factorization for recommender systems," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017. doi: 10.1109/IPDPSW.2017.91 pp. 409–418.
- [21] C. Enrique, T. Alexander, C. Héctor, J. G. Francisco, G. Felipe, B. Belén, and A. Diego, "In-memory distributed software solution to improve the performance of recommender systems," *Software: Practice and Experience*, vol. 47, no. 6, pp. 867–889, 2017. doi: 10.1002/spe.2467. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2467>
- [22] D. Kim and B.-J. Yum, "Collaborative filtering based on iterative principal component analysis," *Expert Syst. Appl.*, vol. 28, no. 4, pp. 823–830, May 2005. doi: 10.1016/j.eswa.2004.12.037. [Online]. Available: <http://dx.doi.org/10.1016/j.eswa.2004.12.037>
- [23] R. Salakhutdinov and A. Mnih, "Probabilistic matrix factorization," in *Proceedings of the 20th International Conference on Neural Information Processing Systems*, ser. NIPS'07. USA: Curran Associates Inc., 2007. ISBN 978-1-60560-352-0 pp. 1257–1264. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2981562.2981720>
- [24] A. N. Tikhonov, "Solution of incorrectly formulated problems and the regularization method," *Soviet Mathematics*, vol. 4, p. 1035–1038, 1963.
- [25] S. Funk, "Netflix update: Try this at home," 2006, available at <http://sifter.org/~simon/journal/20061211.html>.
- [26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1863103.1863113>