Universidade Federal Fluminense

Dânia Naomi Osato Meira

A Comparative Study of Scalable Implementations of the Alternating Least Squares Algorithm for Collaborative Filtering Recommendation

NITERÓI

2015

Universidade Federal Fluminense

Dânia Naomi Osato Meira

A Comparative Study of Scalable Implementations of the Alternating Least Squares Algorithm for Collaborative Filtering Recommendation

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Artificial Intelligence.

Advisor:

Prof. D.Sc. José Viterbo Filho

NITERÓI

2015

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

M514 Meira, Dânia Naomi Osato

A comparative study of scalable implementations of the alternating least squares algorithm for collaborative filtering recommendation / Dânia Naomi Osato Meira. – Niterói, RJ : [s.n.], 2015.

67 f.

Dissertação (Mestrado em Computação) – Universidade Federal Fluminense, 2015.

Orientador: José Viterbo Filho.

1. Algoritmo computacional. 2. Sistema de recomendação. 3. Método dos mínimos quadrados alternantes. 4. Aplicação web. I. Título

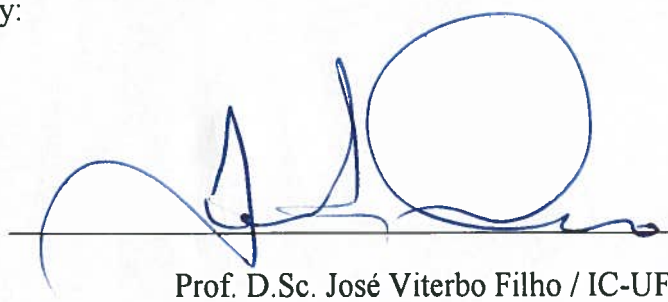
CDD 005.136

A Comparative Study of Scalable Implementations of the Alternating Least Squares Algorithm for Collaborative Filtering Recommendation

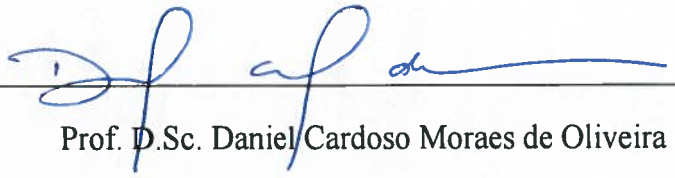
Dânia Naomi Osato Meira

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Artificial Intelligence.

Approved by:



Prof. D.Sc. José Viterbo Filho / IC-UFF (Advisor)



Prof. D.Sc. Daniel Cardoso Moraes de Oliveira / IC-UFF

Prof[a]. D.Sc. Karin Koogan Breitman / EMC – Brazil R&D Center

Niterói, August 13th, 2015.

To my family and friends

Acknowledgments

I would like to express my gratitude for the willingness, support and guidance of my advisor Prof. José Viterbo Filho.

A special thought goes out to all my co-workers at hypermindR. Thank you for the convivence and life lessons.

A great thank goes to my parents for the support they provided me through my entire life. And because even not being here with me every day, their love and support are always present.

I must acknowledge my best friends, Samira Marques, Tarsila Tavares and Charles Kilesse for providing me with such wonderful moments, helping me throughout writing this thesis and my life in general.

Most importantly I want to deeply thank Felipe Cançado, without whose love, encouragement and patience I would not have got this far.

Resumo

Sistemas de recomendação são uma popular ferramenta para sugerir produtos, serviços e informação para potenciais consumidores, que se baseiam em transações passadas e *feedback* de outros usuários têm interesses comuns. Com o enorme crescimento de usuários, produtos e informações disponíveis na internet e o rápido surgimento de novos serviços *online*, a tarefa de gerar diversas recomendações por segundo, para milhares de usuários se tornou uma necessidade e um desafio. Muitos sistemas de recomendação sugerem itens para os usuários utilizando técnicas de filtro colaborativo, que processam o histórico de itens vistos, comprados ou avaliados pelos usuários. Dois principais problemas enfrentados pela maior parte das abordagens de filtro colaborativo são a escalabilidade e a esparsidade da matriz de perfil dos usuários, que foram superados com sucesso pela técnica de modelos de fatores latentes. As concepções mais bem sucedidas de modelos de fatores latentes são baseadas em fatoração de matrizes. Dentre os algoritmos de fatoração de matrizes, o mínimos quadrados alternantes (ALS, do inglês *alternating least squares*), se destaca pelo fato de que suas etapas de cálculo são facilmente paralelizáveis. Neste trabalho propomos uma metodologia para comparação do desempenho entre duas implementações paralelas do algoritmo ALS, um executado com o paradigma MapReduce no arcabouço Apache Hadoop, e outro executado no arcabouço Apache Spark, que faz uso de blocos de forma cuidadosa para reduzir o processo de coleta de lixo da JVM e também para melhor utilizar operações de álgebra linear de alto nível. São realizados experimentos de avaliação de desempenho quanto à acurácia das recomendações e ao tempo de treinamento de ambos os algoritmos quando executados em diferentes conjuntos de dados publicamente disponíveis, de diferentes tamanhos e pertencentes a diferentes domínios de recomendação. Os resultados experimentais confirmam que a implementação no Spark é mais eficiente, uma vez que o processamento é realizado em memória e não em disco como no Hadoop.

Palavras-chave: sistemas de recomendação escaláveis, filtro colaborativo, fatoração de matrizes, alternating least squares, Apache Spark, Apache Hadoop, MLlib, Mahout.

Abstract

Recommender systems are now a popular tool used to suggest products, services and information to potential consumers, based on their profile of past transactions and feedback from other users that share similar interests. With the tremendous growth of users, products and information made available on the web and the rapid introduction of new e-business services, performing many recommendations per second for millions of users has become a necessity and a challenge. Many recommender systems suggest items to users employing collaborative filtering techniques, which process historical records of items that the users have viewed, purchased, or rated. Two major problems that most collaborative filtering approaches have to resolve are scalability and sparseness of the user's profile matrix, which have been successfully overcome with the use of latent factor models technique. 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 because its computations are easily parallelizable. 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, which makes careful use of blocking to reduce JVM garbage collection overhead and to utilize higher-level linear algebra operations. We perform 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.

Keywords: scalable recommender systems, collaborative filtering, matrix factorization, alternating least squares, Apache Spark, Apache Hadoop, MLlib, Mahout.

List of Figures

List of Tables

Contents

Chapter 1 Introduction

Given the increasing amount and growing variety of products, services and information, which are daily made available on the Web, and the rapid introduction of new e-business services, making a choice from such wide range of options can be somewhat complex and difficult to manage. It can be argued that, while being able to choose is good, having so many options to choose from is not always more rewarding. Indeed, choice, with its implications of freedom, autonomy, and self-determination can become excessive, causing a sense that freedom may come to be regarded as a kind of misery-inducing tyranny [1].

To prevent overwhelmed users from making poor decisions, recommender systems came into focus, facilitating users' access to information about the items they are most likely to be interested, whether such items are books, movies, music, videos, Web pages, news or services, among others. Recommender systems provide users an efficient way for reviewing their options and selecting the most suitable items, since attempting to infer their preference and recommend new items that maximizes the probability of fitting each of the users' preferences.

Recommender systems have been applied in diverse domains such as e-commerce, digital libraries, search engines and news websites. Netflix [2], one of the largest video streaming service, which started in the United States and is now available worldwide, is a successful example of the use of this powerful tool. In 2006, the company launched a challenge called Netflix Prize (Netflix Prize), offering a prize of one million dollars to anyone who could get a 10% improvement in the level of accuracy of its movie recommender system for customers. Thousands of participating teams, from the entire world, proposed new and better solutions. In September 2009, the BellKor's Pragmatic Chaos team was declared the winner [3].

A recommender system is designed to suggest items individually for each user based on his long-term and/or shot-term preferences [1]. The purpose is to enlighten the user about the items that should be of his interest, amongst many different options. User preferences can be collected implicitly or explicitly [4]. In the implicit form, information is obtained through past purchases, history of visited Websites, clicked links, browser cookies or even geographical location options, for instance. The explicit form of collecting preferences consists in using effective feedback, such as ratings given to particular items. In both cases, the users' preferences are stored in a user-item matrix referred to as ratings matrix.

Among the many techniques used to make recommendations, this work focuses on collaborative filtering, which is based on comparing the profile of preferences of the users. This

technique is very popular in e-commerce applications, due to its good results [5]. The methods that can be used as collaborative filtering are divided into two general classes: neighborhood-based and model-based.

The neighborhood-based methods, known as item-based or user-based collaborative filtering, make use of the neighborhood relationship among items and users to discover new items for recommendation, and may be seen as the analogue of nearest-neighbor classifiers. In the item-based collaborative filtering, the neighborhood is defined as a set of items that have similar ratings when evaluated by several users of the system. Then, the system is able to predict the rating of a user for a specific item based on the ratings from this user for items in the neighborhood of such item [6]. In the user-based collaborative filtering the neighborhood is composed of users who gave similar ratings to the same items. The idea is to group users that have similar interests, because an item that user A likes is potentially interesting for user B if they showed similar interests in the past.

This approach presents serious 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, it may not be appropriate for online systems which recommend in real time. Furthermore, these algorithms are more sensitive than the model-based to some common problems of recommender systems.

One common problem is the sparsity of the rating matrix, which 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 [7]. For instance, the recommendation generated by Spotify, a digital music platform [8], is based on history of streamed tracks, so it groups users who have listened to similar songs. When such systems have access only to a small number of past streamed records relative to the total numbers of the songs and users, determining which users are similar to each other and what their interests are becomes difficult.

Another recurrent problem in generating recommendations happen 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 [9]. However, there are models can help bridge the gap from existing items to new items, by inferring similarities among them. One of these models is the ones based on matrix factorization, as discussed next.

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 [10], neural networks [11], and Singular Value Decomposition [12] [13] [14]. 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 [15].

These problems become more evident when trying to construct recommender systems associated with Websites that have a large number of users and items 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 [8] 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 [16]. On the news field, Globo.com Website [17] receives more than 10 million visitors per day, with a rate of 32 thousand accesses per minute [18]. Amazon [19] generates recommendations from a database with 253 million products [20] for users of 270 million active customer accounts [21]. Popular websites in the social media field also impose challenges to recommender systems: twitter.com [22] had 236 million active users per month in the first quarter of 2015 [23] and at the same period, facebook.com [24] had a monthly active base of around 1.44 billion users [25].

An efficient approach is essential in all those cases. Nowadays, to tackle such performance challenges, online recommender systems have combined two strategies: efficient algorithms, that avoid the computational complexity of calculating each of the entries of the high dimensional and sparse matrix; and 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 [26]. 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 [27]. 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 [28].

Recent works suggested modeling only the observed ratings, while avoiding overfitting, through an adequate regularized model [29] [30] [31]. Some parallel algorithms for latent factor models with regularization have been designed aiming at improving the modelling performance. Among them, two can be highlighted: the low-rank matrix factorization with Alternating Least Squares (ALS), which uses a series of broadcast-joins [32], built on top of the open source MapReduce implementation Hadoop [33] and its ecosystem, and the Alternating Least Squares with Weighted-λ-Regularization (ALS-WR) [34] which has been implemented in Apache Spark's Machine Learning library, MLlib [35].

The Alternating Least Squares is an iterative algorithm that models the rating matrix as the multiplication of two low-rank factor matrices: one represents the users and another represents the items. At each iteration, the algorithm fixes one matrix and learns its factors by solving a least-squares minimization problem, which means minimizing the reconstruction error of the observed ratings. Then it alternates, to refine the other matrix, and this continues until it converges. To begin the process, one matrix is filled with random feature vectors. This characterizes ALS computations as parallelizable, once they can be decomposed into straightforward linear algebra operations on each row independently, leading to highly scalable algorithms [36].

1.1 Problem Definition

Scalability and performance are key issues for recommender systems, since computational complexity increases with the number of users and items. Despite the fact that the mentioned implementations have offered some evolution regarding the efficiency of recommendation algorithms, the MapReduce paradigm does not offer the best performance for machine learning algorithms, as it demands a great amount of iterations involving writing and reading data from disk [37]. To improve efficiency, Apache Spark [38] has emerged as a fast and general engine for big data that processes data in-memory. Spark provides implementations of some machine learning algorithms in a specific library called MLlib [39], including the Alternating Least Squares algorithm for collaborative filtering recommendation, but the performance gain for this implementation has not yet been systematically evaluated in any comparative study.

1.2 Goals

The goal of this work is to provide a comparative study of two different scalable implementations of the Alternating Least Squares algorithm for collaborative filtering recommendation: the Hadoop-based Mahout ALS implementation and the Spark-based MLlib ALS

Implementation. For that purpose, we first describe in details the alternating least squares with weighted regularization algorithm, that is parallelized and optimized to scale up well with large, sparse data which makes it a strong candidate when computing recommendations in real time. Then, we compare the two implementations of this algorithm on a Linux cluster as the experimental platform, to assess the performance improvement made available through Apache Spark, when compared to the Hadoop MapReduce framework. The experimental results are obtained using two different datasets: Jester and MovieLens. For both datasets, we prove Spark's beneficial use when setting up scalable collaborative filtering recommender systems, providing an evaluation of the performance gain when working in two different recommendation domains with large collections of data.

1.3 Contributions

The contributions of this master thesis are the following: (1) The description of a methodology for comparing different implementations of machine learning algorithms. (2) An experimental evaluation of two different scalable implementations of the ALS algorithm for collaborative filtering recommender systems, Mahout ALS and the MLlib ALS, using two different datasets and assessing efficiency, accuracy and scalability aspects of recommendation problems. (3) A demonstration that the distributed and parallel approach is suitable for real-world use cases.

1.4 Organization

This master thesis is composed by seven more chapters, organized as follows:

- *Chapter 2*: *Recommender Systems*. In this chapter, we explain the fundamental concepts about recommender systems and its main approaches.
- *Chapter 3: Big Data Technologies*. In this chapter, we present the concepts and history of some of the emerging big data technologies that are related to the topic of this master thesis — projects Apache Hadoop and Apache Spark.
- *Chapter 4: Related Work*. In this chapter, we describe some research related to the topic of this master thesis, explaining how they improve efficiency of recommender algorithms by utilizing the technologies described in the previous chapter.
- *Chapter 5: Comparative Study*. In this chapter, 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 dimensions: accuracy, efficiency and scalability.

- *Chapter 6: Conclusion*. In this chapter, we discuss the contributions and limitations of the proposed study, presenting also some topics for future work.

Chapter 2 Recommender Systems

In this chapter we present the main concepts of recommender systems, fundamental for the understanding of this work. In Section 2.1 we describe the most general idea of recommender systems, and next we discuss its two main approaches: Content-based Filtering in Section 2.2 and, Collaborative Filtering (CF) in Section 2.3. Our focus here is to describe the general recommender systems setting as a base for the investigated algorithm in this work, the Alternating Least Squares (ALS).

2.1 Concept

Recommender Systems are a set of techniques providing suggestions for items to be of use to a user. The suggestions relate to various decision-making processes, such as which items to buy, what music to listen to, or what online news to read. "Item" is the general term used to denote what the system recommends to the users.

In their simplest form, personalized recommendations are offered as ranked lists of items, as represented in Figure 2-1: machine learning algorithms are trained in order to predict what are the most suitable items, based on the user's preferences and constraints [1].

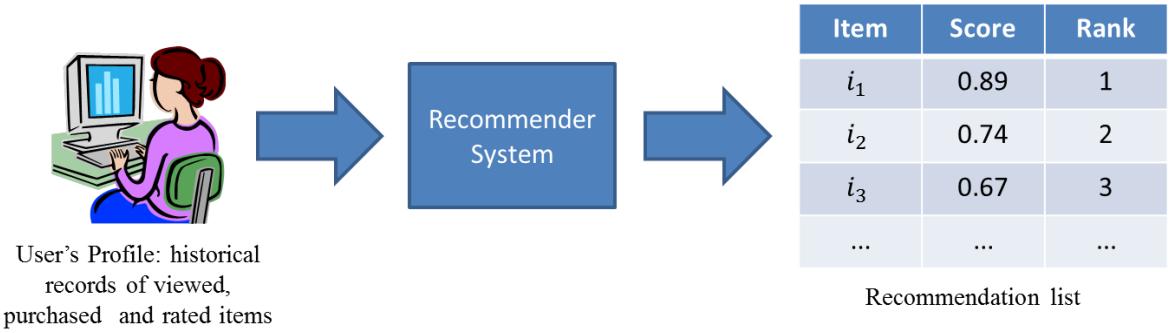


Figure 2-1: Personalized Recommendations

There are two general different approaches to score and recommend items, and each of them analyzes different kinds of data to develop notions of affinity between users and items. Content-based Filtering systems are based on user's profile attributes, while Collaborative Filtering systems analyze historical interactions. Hybrid systems are the ones that combine both approaches [4]. Figure 2-2 resumes the approaches to solve the recommendation problem.

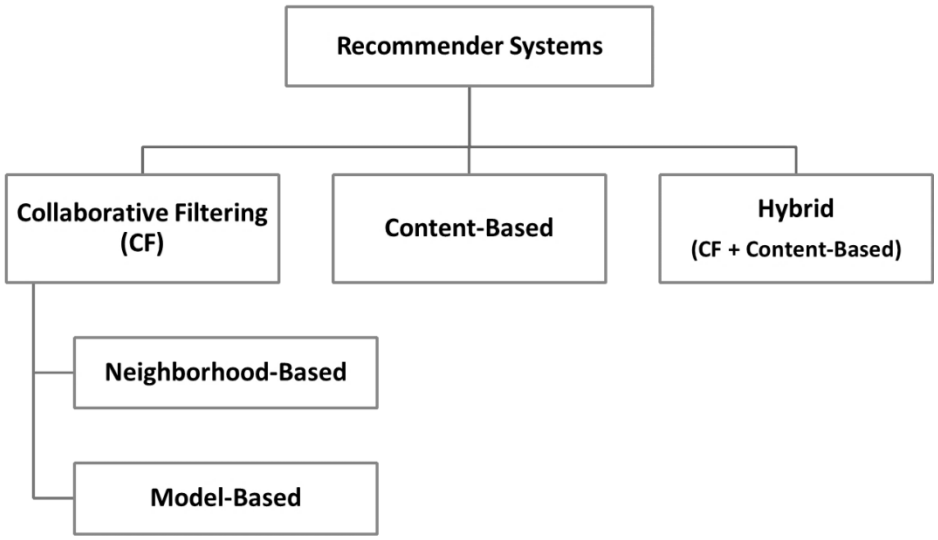


Figure 2-2: Taxonomy of recommender systems approaches

2.2 Content-based Filtering

As the name implies, this approach exploits the content of items, *i.e.*, their peculiar features, to predict its relevance for the user. The content-based approach analyzes a set of descriptions of the items previously rated by a user, and from this input builds a user profile, which is a structured representation of the user's interests. The process to find new interesting items consists in matching up the attributes highlighted in the user profile against the attributes that describe the content of an item. The result is a relevance judgment that tries to represent the user's level of interest in that object. A schematic of this approach is found on Figure 2-3.

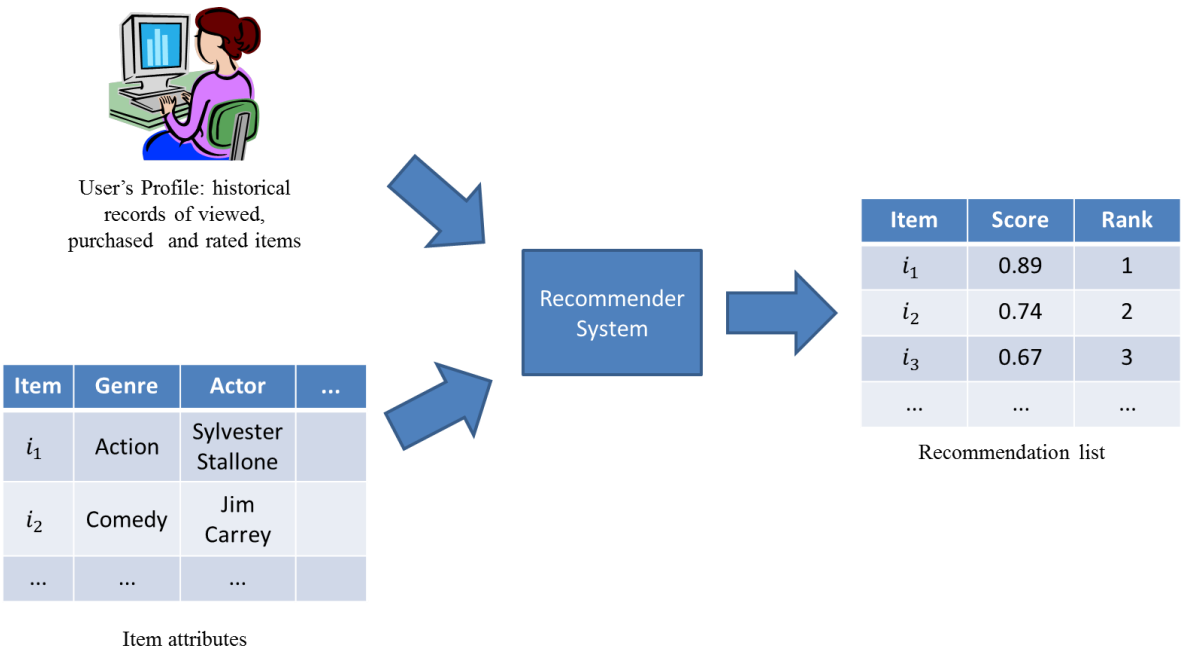


Figure 2-3: Schematic view of the content-based recommender system approach

Research on content-based recommender systems takes place at the intersection with the topic of Information Retrieval (IR). From the IR recommendation technologies research derives the vision that users searching for recommendations are engaged in an information seeking process. The difference is that instead of giving a query to search for, usually a list of keywords, the user is represented by his/her own profile of interests. Also, the use of document modeling techniques with roots in IR is very desirable when the items to be recommended are web pages, news, emails or documents because they are described through unstructured text and its attributes can be very different, that is, attributes with undefined and non-standardized values.

2.3 Collaborative Filtering (CF)

The basic idea of collaborative filtering-based algorithms is to provide item recommendations or predictions based on the opinions of other similar users, as shown in Figure 2-4. The opinions of users can be obtained explicitly, when the users rate items (*e.g.* in a scale from 1 to 5), or through implicit feedback, when the system can infer user preferences by observing user behavior, which indirectly reflects opinion [36]. Types of implicit feedback include purchase history, browsing history, search patterns, or even mouse movements, which are mapped into a set of implicit ratings: *e.g.* purchase of item is represented by rating 5, add item to cart equals rating 4 and click is a rating 3.

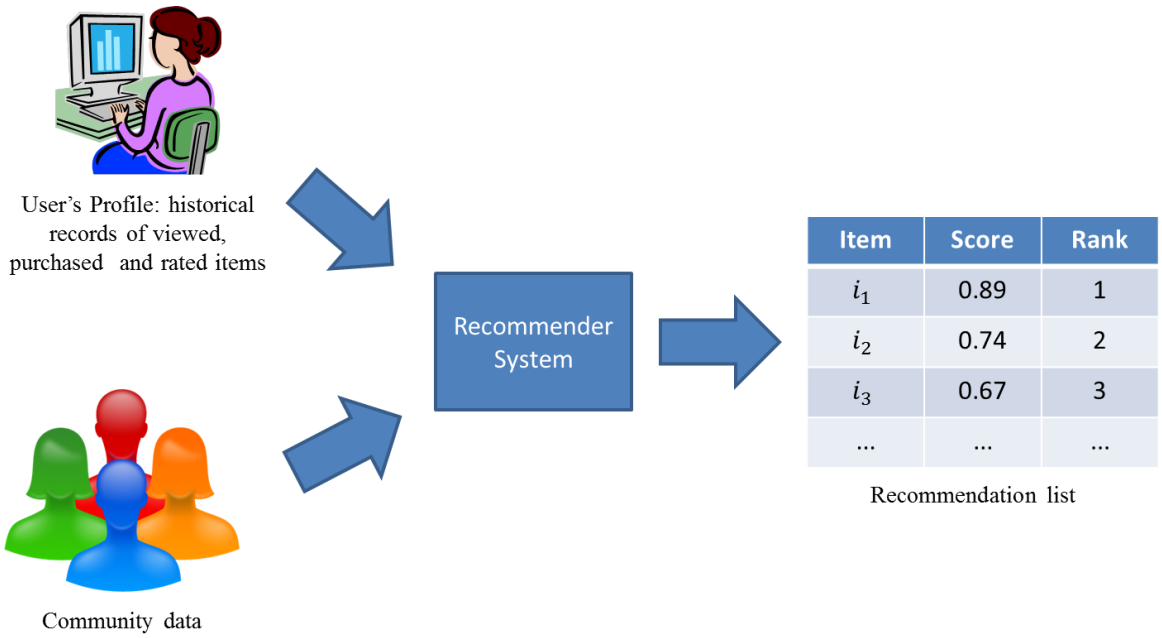


Figure 2-4: Schematic view of the collaborative filtering recommender system approach

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. Figure 2-5 shows an example of a ratings matrix where predictions are being computed for movies.

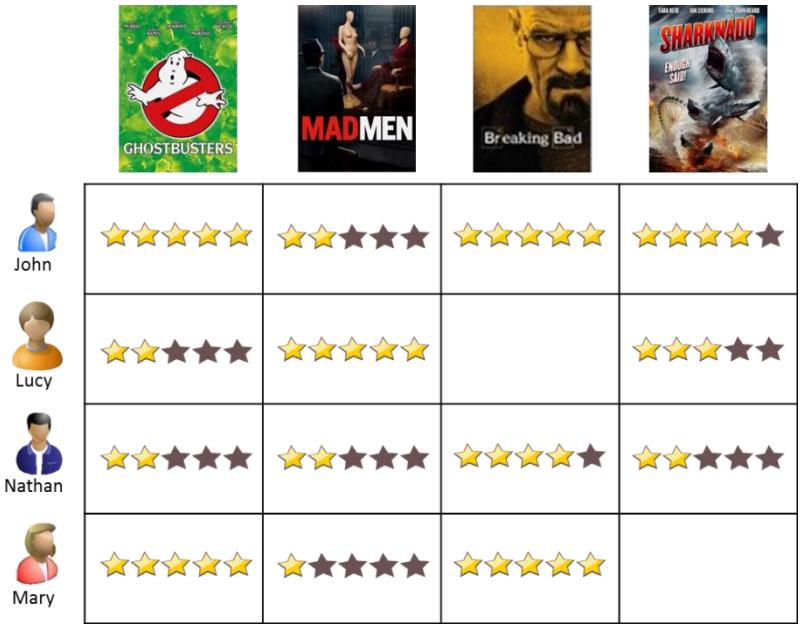


Figure 2-5: Ratings Matrix Example

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 uncover 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.

2.3.1 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 [34]. A representation of the matrix factorization technique is given in Figure 2-6:

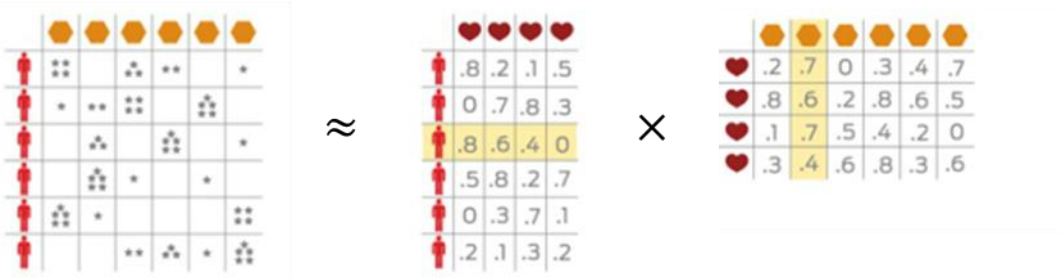


Figure 2-6: Scheme of the matrix factorization technique

For example, when the items are movies, factors might measure obvious dimensions such as comedy vs. drama, as illustrated on Figure 2-7.

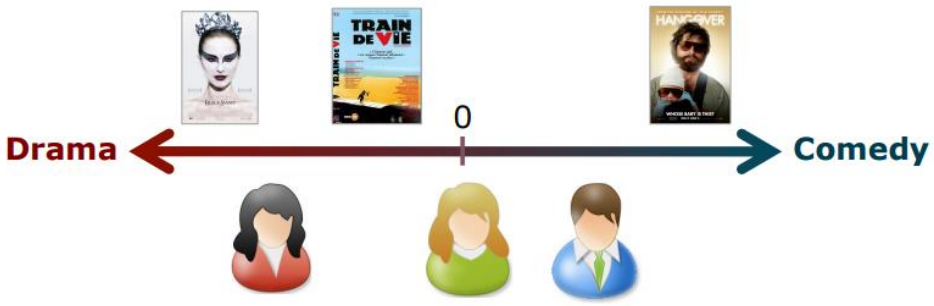


Figure 2-7: Latent Factor Example - Movie Genre

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 results in best recommendations (best summarization of the concepts) for a 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, $\widehat{R}$ 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 $q_i \in \mathbb{R}^k$, user-item interactions are modeled as inner products:

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

An example of matrix factorization computation is found next on Figure 2-8. On a system with five users and six items, 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,* q_3^T and *user 4* , p_4.

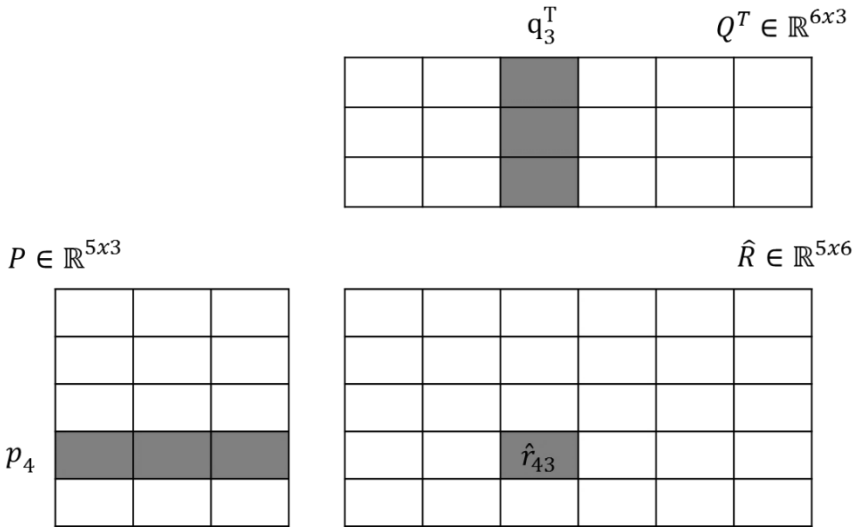


Figure 2-8: Example of a matrix factorization computation

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, since in general users have rated only a small set of items [28]. The SVD can be computed one column at a time, whereas for the partially specified case, no such recursive formulation holds [34]. Also, addressing only few known ratings is highly likely to model overfitting [26]. Earlier works relied on imputation [12] [40], 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 [41] [42] [13] [43] [36]. This model minimizes the regularized squared error on the set of observed ratings:

$$\min_{p,q} \sum_{(u,i)\epsilon\kappa} \left(r_{ui} - q_i^T p_u\right)^2 + \lambda(\|q_i\|^2 + \|p_u\|^2) \quad (2)$$

where κ 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 λ constant [44]. This is also known as the Tikhonov regularization [45].

Eq. (2) is solved using a learning algorithm such as stochastic gradient descent (SGD) [42] or alternating least squares (ALS) [34].

2.3.2 Stochastic Gradient Descent (SGD)

Also known as incremental or online learning, the stochastic gradient descent (SGD) algorithm loops through all ratings in the training set [46]. For each given training case, the system predicts r_{ui} and computes the associated prediction error:

$$e_{ui} \stackrel{\text{def}}{=} r_{ui} - q_i^T p_u$$

Then it modifies the parameters by a magnitude proportional to γ in the opposite direction of the gradient, yielding:

$$q_i \leftarrow q_i + \gamma \cdot (e_{ui} \cdot p_u - \lambda \cdot q_i)$$
$$p_u \leftarrow p_u + \gamma \cdot (e_{ui} \cdot q_i - \lambda \cdot p_u)$$

For large-scale problems such as the Netflix Prize, calculating the gradient is computationally expensive, so the SGD algorithm approximates the gradient by taking the gradient at a single observation. One might wonder why replacing exact gradients (GD) by noisy estimates (SGD) can be beneficial. The main reason is that exact gradient computation is costly, whereas noisy estimates are quick and easy to obtain. In a given amount of time, we can perform many quick-and-dirty SGD updates instead of a few, carefully planned GD steps. The noisy process also helps in escaping local minima (especially those with a small basin of attraction and more so in the beginning, when the step sizes are large). Moreover, SGD is able to exploit repetition within the data. Parameter updates based on data from a certain row or column will also decrease the loss in similar rows and columns. Thus the more similarity there is, the better SGD performs. Ultimately, the hope is that the increased number of steps leads to faster convergence [47].

2.3.3 Alternating Least Squares (ALS)

Since both q_i and p_u are unknown, Eq. (2) is not convex. 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 $p_u's$'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 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 [36].

The second case is when the system can use parallelization. The algorithm computes each $\mathrm{q_i}$ independently of the other item factors and computes each $\mathrm{p_u}$ independently of the other user factors, which allows for massive parallelization of the implementation [34].

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.

The algorithm can be summarized in the following steps:

Step 1 Initialize matrix Q to a random value;

Step 2 Fix Q, solve P by minimizing the objective function (the sum of squared errors);

Step 3 Fix P, solve Q by minimizing the objective function similarly;

Step 4 Repeat Steps 2 and 3 until a stopping criterion is satisfied.

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.

Here follows a demonstration of how to solve $\mathrm{p_u}'$s's when $\mathrm{q_i}'$s are fixed.

Each $\mathrm{p_u}$ is determined by solving a regularized linear least squares problem involving the known ratings of user u, and the feature vectors $\mathrm{q_i}$ of the items that user u has rated:

$$\frac{1}{2}\frac{\partial f}{\partial p_{ju}} = 0 \; \forall j, u$$

$$\Rightarrow \sum_{i \in I_u} (p_u^{\mathrm{T}} q_{\mathrm{i}} - \mathrm{r_{ui}}) q_{ji} + \lambda n_{m_u} p_{ju} = 0 \;\; \forall j, u$$

$$\Rightarrow \sum_{i \in I_u} \mathrm{q}_{ji} p_u^{\mathrm{T}} q_{\mathrm{i}} + \lambda n_{m_u} p_{ju} = \mathrm{r_{ui}}\, \mathrm{q}_{ji} \; \forall j, u$$

$$\Rightarrow \left(Q_{I_u}^T Q_{I_u} + \lambda n_{m_u} E\right) p_u = Q_{I_u} R^T(u, I_u) \; \forall u$$

$$\Rightarrow \mathrm{p_u} = \; A_u^{-1} V_u \; \forall u$$

Where:

$$A_u = Q_{I_u}^T Q_{I_u} + \lambda n_{m_u} E$$

$$V_u = Q_{I_u} R^T(u, I_u)$$

And E is the $\kappa \times \kappa$ identity matrix. $\mathrm{Q_{I_u}}$ denotes the sub-matrix of Q where columns $\mathrm{i} \in \mathrm{I_u}$ are selected, and $\mathrm{R^T(u, I_u)}$ is the row vector where columns $\mathrm{i} \in \mathrm{I_u}$ of the u-th row of R is taken.

Similarly, when $\mathrm{p_u}'\mathrm{s}$ are fixed, it is possible to compute individual $\mathrm{q_i}$'s via a regularized linear least squares solution, using the feture vectors of users who rated item i, and their ratings of it, as follows:

$$\mathrm{q_i} = \; \mathrm{A_i^{-1} V_i} \; \forall \mathrm{i}$$

Where:

$$\mathrm{A_i} = \mathrm{P_{I_i}^T P_{I_i}} + \lambda \mathrm{n_{m_i} E}$$

$$\mathrm{V_i} = \mathrm{P_{I_i} R^T(i, I_i)}$$

And $\mathrm{P_{I_i}}$ denotes the sub-matrix of P where columns $\mathrm{u} \in \mathrm{I_i}$ are selected, and $\mathrm{R^T(I_i, i)}$ is the row vector where columns $\mathrm{u} \in \mathrm{I_i}$ of the i-th row of R is taken.

Chapter 3 Big Data Technologies

In this chapter we present Apache Hadoop and Apache Spark, the two big data technologies used in this work to build scalable recommender systems. Since both implement the alternating least squares collaborative filtering algorithm on their own machine learning library, the motivation for this discussion is to understand each of the frameworks proposal, the functionality of its modules, and the concepts which we will refer throughout this work, regarding the different implementations and experiments results.

3.1 Apache Hadoop

The Apache Hadoop software library [48] is a system for distributed computing of large datasets among commodity servers. It is an open-source project that brings the ability to cheaply process large amounts of data, regardless of its structure, and it is designed to scale up from single servers to thousands of machines, each offering local computation and storage. The release 1.2.1 of the project, which is the one used in this work, includes the modules:

- Hadoop Common: the common utilities that support the other Hadoop modules;
- Hadoop Distributed File System (HDFS): a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- Hadoop MapReduce: A framework for parallel processing of large data sets on large clusters of commodity hardware in a reliable, fault-tolerant manner.

There are also many Hadoop-based solutions, such as Pig: a high-level data-flow language and execution framework for parallel computation [49]; Hive: a data warehouse infrastructure that provides data summarization and ad hoc querying [50]; HBase: a scalable, distributed database that supports structured data storage for large tables [51] and Mahout: a scalable machine learning and data mining library [52].

3.1.1 Hadoop Distributed File System (HDFS)

HDFS is the primary distributed storage used by Hadoop applications. It is a distributed file system with high fault-tolerance, and it is designed to be deployed in low-cost software [53].

A HDFS has a master/slave architecture. A HDFS cluster consists of a single namenode, a master server that manages the filesystem metadata and a number of datanodes, usually one per node, that store part of the actual data. Since applications that run on HDFS have large data sets, typically gigabytes to terabytes in size, it should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. Each of these nodes has a non-trivial probability of failure, meaning that

some component of HDFS is always non-functional. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. Also, it has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications.

3.1.2 Hadoop MapReduce

MapReduce is a functional paradigm for data-intensive parallel processing on clusters [54]. A MapReduce job usually splits the input dataset into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps grouped by their key, which are then sent as input to the reduce tasks in the shuffle phase. The receiving machines merge the tuples, that is, invoke the reduce function on all tuples sharing the same key, and finally write the output. The shuffle phase is typically the most costly operation as the mappers' output is spilled to disk at first and each reducer downloads its assigned data from every mapper afterwards.

Typically both the input and the output of the job are stored in a file-system and the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the HDFS are running on the same set of nodes, as shows Figure 3-1. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

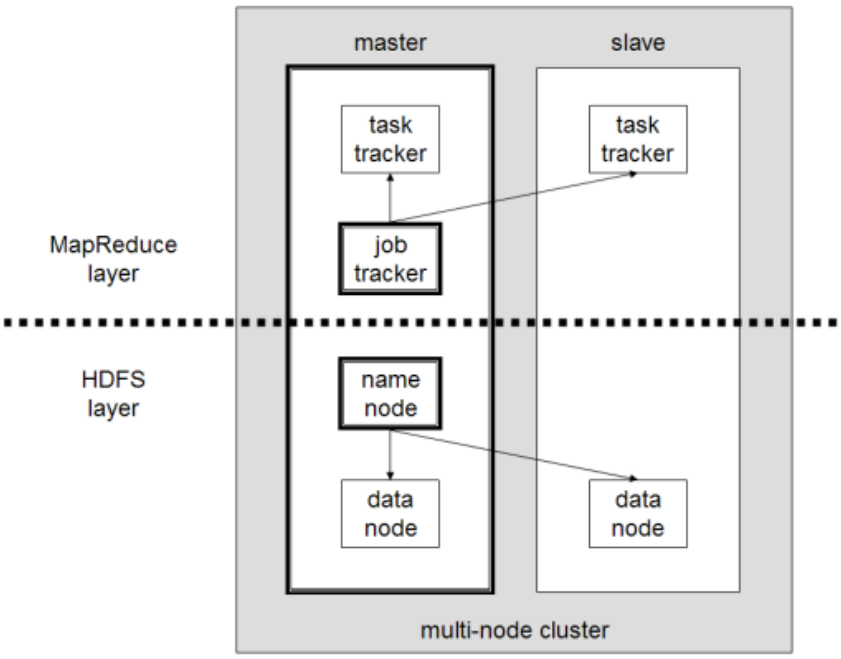


Figure 3-1: A multi-node Hadoop Cluster [55]

This model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. It achieves scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention. While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows [56].

3.1.3 Mahout

Apache Mahout [52] is open-source library that implements distributed and scalable machine learning and data mining algorithms. The algorithms are in the areas of collaborative filtering, clustering and classification. Many of the implementations until the 0.9 release runs on top of Hadoop environment with the support of MapReduce and HDFS, but Mahout also provides Java libraries for common math operations (focused on linear algebra and statistics) and primitive Java collections. Mahout has a map-reduce implementation of ALS, which will be discussed in details on Chapter 5.

In April of 2015 Mahout released a new version (0.10.0) which introduced a new math environment called Samsara, a Linear Algebra library for Mahout. It is a new generation of Mahout that entirely re-thinks philosophy of previous Mahout line of releases.

Samsara is written in Scala, which makes it possible to use operator overloading and it features nice R-like or Matlab-like syntax for basic Linear Algebra operations [57]. For example, matrix multiplication is just X %*% Y. Also, these operations can be distributed and run by an executing environment, currently by Apache Spark.

This is a major change in the project's direction, since it abandons MapReduce based algorithms. The existent algorithms are still supported but from this release on it will not accept new MapReduce implementations [58].

3.2 Apache Spark

Apache Spark [59] is a fast and general engine for large-scale data processing. It is a popular open-source platform for large-scale data processing that is well-suited for iterative machine learning tasks. It was originally developed in 2009 in UC Berkeley's AMPLab [60], and open sourced in 2010. After being released, Spark grew a developer community on GitHub and moved to Apache in 2013 and today a wide range of contributors to the project, over 400 developers from 100 companies [61].

In general, the Hadoop framework is a poor fit for iterative algorithms, since every iteration has to be scheduled as a single MapReduce Job with a high startup cost (potentially up to 10 seconds). Further, the system creates a lot of unnecessary I/O and network traffic as all static, iteration-invariant data has to be re-read from disk and re-processed during each iteration and the intermediary result of each iteration has to be materialized in HDFS [32]. Spark research is motivated by improvement of applications that reuse a working dataset across multiple parallel operations, which includes many iterative machine learning algorithms and interactive data analysis tools, which Hadoop users report that MapReduce is deficient [56].

To use Spark, developers write a driver program that implements the high-level control flow of their application and launches various operations in parallel. Spark provides two main abstractions for parallel programming: resilient distributed datasets (RDD) and parallel operations on these datasets.

Spark provides high-level APIs in Java, Scala and Python, and an optimized engine that supports general execution graphs. Spark provides APIs supporting related tasks, like data access, ETL, and integration.

Spark supports a rich set of higher-level tools, illustrated on Figure 3-2, which includes Spark SQL for SQL and structured data processing, Spark Streaming for processing of live data streams, MLlib for machine learning and GraphX for graph processing.

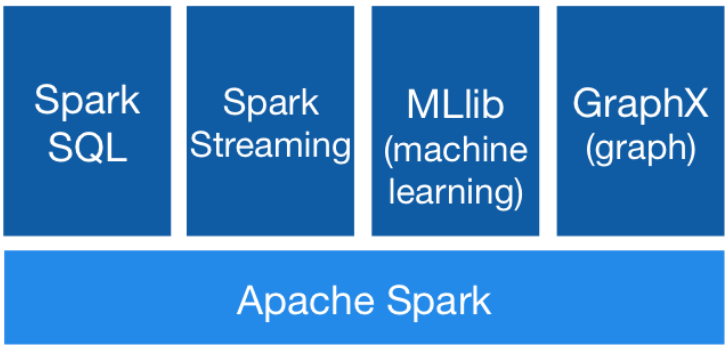


Figure 3-2: Spark built-in libraries [59]

3.2.1 Resilient Distributed Dataset (RDD)

Spark primary abstraction is the Resilient Distributed Dataset (RDD): a fault-tolerant abstraction for in-memory cluster computing, which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition [56].

Formally, an RDD is a read-only, partitioned collection of records and by default, RDDs are lazy and ephemeral, which means that partitions of a dataset are materialized on demand when they are used in a parallel operation, and are discarded from memory after use. RDDs can only be created through deterministic operations on either data in stable storage (e.g. HDFS) or other RDDs by changing the persistence of an existing RDD, when caching or saving it or by transforming an existing RDD.

The greatest strength of Spark is the ability to cache RDDs in memory. This allows to run iterative algorithms faster than using the typical Hadoop based MapReduce framework [38].

3.2.2 Spark SQL

Spark SQL allows relational queries expressed in SQL, HiveQL, or Scala to be executed using Spark. At the core of this component is a new type of RDD, named SchemaRDD. SchemaRDD is composed of Row objects, along with a schema that describes the data types of each column in the row. A SchemaRDD is similar to a table in a traditional relational database and it can be created from an existing RDD, a Parquet file, a JSON dataset, or by running HiveQL against data stored in Apache Hive [62].

3.2.3 Spark Streaming

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches. The processed data can be pushed out to filesystems, databases, and live dashboards, but it is also possible to apply machine learning algorithms available at MLlib and graph processing algorithms available at GraphX on these data streams [63]. The Spark Streaming workflow is illustrated on Figure 3-3:

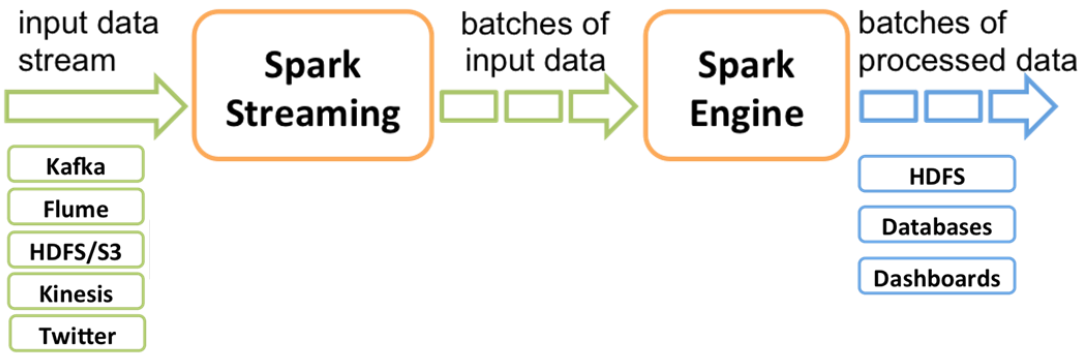


Figure 3-3: Spark Streaming workflow [63]

3.2.4 MLlib

The scalable a machine-learning library, MLlib [35], consists of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives. Spark excels at iterative computation, enabling MLlib to run fast, making possible to yield better results than the one-pass approximations sometimes used on MapReduce.

MLlib features a blocked implementation of the ALS algorithm that leverages Spark's efficient support for distributed, iterative computation. It uses native LAPACK (Linear Algebra Package) [64], a standard software library for numerical linear algebra, to achieve high performance and scales to billions of ratings on commodity clusters. This implementation is discussed in details on Chapter 5.

3.2.5 GraphX

GraphX is a Spark API for graphs and graph-parallel computation. At a high-level, GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge. To support graph computation, GraphX

exposes a set of fundamental operators (*e.g.*, subgraph, joinVertices, and mapReduceTriplets) as well as an optimized variant of the Pregel API. In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

The goal of the GraphX project is to unify graph-parallel and data-parallel computation in one system with a single composable API. The GraphX API enables users to view data both as a graph and as collections (*i.e.*, RDDs) without data movement or duplication. By incorporating recent advances in graph-parallel systems, GraphX is able to optimize the execution of graph operations [65].

Chapter 4 Related Work

In this chapter, we discuss previous works that propose matrix factorization implementations to solve the recommendation problem. We will describe each of the proposed implementations and the performance results obtained. On section 4.1 we focus on the work that established the matrix factorization technique as a dominant methodology within collaborative filtering recommenders, as experiments with datasets such as the Netflix Prize data showed that it delivers accuracy superior to classical nearest-neighbor techniques and offering at the same time a compact memory-efficient model that systems can learn relatively easily [28]. On section 4.2 we discuss two publications that proposed parallel implementations of the ALS algorithm, as this is the approach underlying both Mahout and MLlib implementations. The first introduces the regularization parameter [34] and is implemented in Matlab [66], and the second uses broadcasts-joins executed on MapReduce [32]. Finally, on section 4.3 we present the work that proposes a method to execute the same ALS algorithm on implicit feedback data, which is a very common situation [36].

4.1 Traditional Matrix Factorization

The Netflix Prize [67] competition that began in October 2006 has motivated the progress in the field of collaborative filtering. For the first time, the research community gained access to a large-scale, industrial strength data set of more than 100 million movie ratings spanning about 500,000 anonymous customers and their ratings on more than 17,000 movies, each movie being rated on a scale of 1 to 5 stars. This was very attractive to thousands of scientists, students, engineers and enthusiasts to the field, the data have been downloaded by more than 41,000 teams from 186 different countries [68].

Participating teams submit predicted ratings for a test set of approximately 3 million ratings, and Netflix calculates a root-mean-square error (RMSE) based on the held-out truth. The first team that can improve on the Netflix algorithm’s RMSE performance by 10 percent or more would win a $1 million prize.

The nature of the competition has encouraged rapid development, where innovators built on each generation of techniques to improve prediction accuracy. Because all methods are judged by the same rigid yardstick on common data, the evolution of more powerful models has been especially efficient.

In September 2009 the prize was awarded to the BellKor’s Pragmatic Chaos team [69] 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 [70].

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 then 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 [71] [72] [73].

The SGD implementation used in this solution is described by Simon Funk [42] 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 the in the paper describing the winning solution it is not 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 time they needed, since the 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 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 [74].

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 [74]. 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 [75] [76].

4.2 Parallel Matrix Factorization

As mentioned on the previous section, the Netflix Prize engaged many researchers on the collaborative filtering field. Another participating team proposed, in 2008, a parallel implementation of matrix factorization, called the Alternating-Least-Squares with Weighted-λ-Regularization (ALS-

WR) [34]. 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.

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.

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):

$$\min_{p,q} \sum_{(u,i)\epsilon\kappa} \left(r_{ui} - q_i^T p_u\right)^2 + \lambda\left(\sum_i n_{m_i} \|q_i\|^2 + \sum_u n_{m_u} \|p_u\|^2\right) \quad (3)$$

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

The solution for Eq. (3) follows the steps demonstrated on section 2.3.2 but instead of initializing the matrix Q to random values on Step 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 U, if the difference between the observed RMSEs is less than 0.0001, the iteration stops and the obtained M, U are used to make final predictions on the test dataset.

In this cited approach, a version that allows for parallel computation of Matlab [66] 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 update 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.

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 [32]. 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 at GitHub [77]. 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 [34].

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 $max(|M|, |N|) * k * 8\, byte$, as alternatingly, 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 [78]. 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 4-1 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.

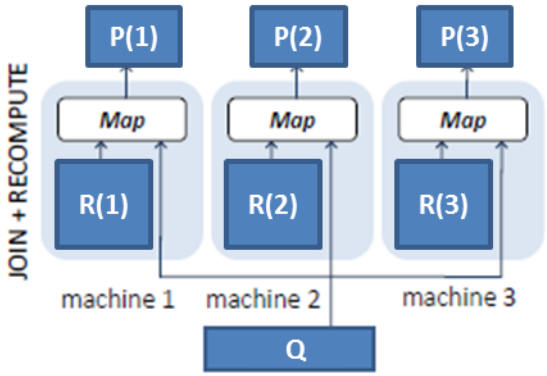


Figure 4-1: Parallel recomputation of user features by a broadcast join [32]

This proposed approach is able to avoid some of the drawbacks of MapReduce and the Hadoop implementation described in Section 3.1.2. 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 DFS. This implementation contains multithreaded mappers that leverage all cores of the worker machines for the re-computation of the feature matrices and uses JBlas [79] 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. An alternative would be to use a repartition join [78] with constant network traffic and linear scale-out, as illustrated on Figure 4-2. However, this argumentation only looks at the required network traffic and does not account for the fact that the computation via repartition-join would also need to sort and materialize the intermediate data in each step. The conclusion is that the approach with a series of broadcast-joins is to be preferred for common production scenarios, since the cluster size would have to exceed several hundred machines to have the broadcast-join technique cause more network traffic than a computation via repartition-join.

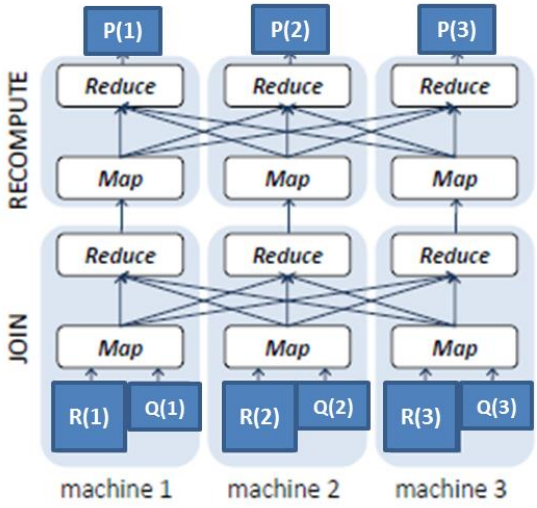


Figure 4-2: Parallel recomputation of user features by a repartition join [32]

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.

4.3 Matrix Factorization for Implicit Feedback Datasets

The work exposed here proposes treating data as indication of positive and negative preference associated with vastly varying confidence levels, leading to a new proposed factor model which is especially tailored for implicit feedback recommenders [36]. Their approach is different from the traditional explicit feedback approach as it modifies both the model formulation and the optimization technique in order to better treat the unique properties of implicit feedback datasets.

Also, it is suggested a scalable optimization procedure, which scales linearly with the data size. The python implementation is available at GitHub [80].

Recommender systems rely on different types of input. Most convenient is the high quality explicit feedback, which includes explicit input by users regarding their interest in products, like Netflix that collects star ratings for movies. However, explicit feedback is not always available but recommenders can infer user preferences from the more abundant implicit feedback, which indirectly reflect opinion through observing user behavior [36]. Types of implicit feedback include purchase history, browsing history, search patterns, or even mouse movements. For example, a user that purchased many books by the same author probably likes that author.

The vast majority of the literature in the field is focused on processing explicit feedback; probably thanks to the convenience of using this kind of pure information. However, in many practical situations recommender systems need to be centered on implicit feedback. This may reflect reluctance of users to rate products, or limitations of the system that is unable to collect explicit feedback. In an implicit model, once the user gives approval to collect usage data, no additional explicit feedback (*e.g.* ratings) is required on the user's part.

Unique characteristics of implicit feedback are listed as follows:

1. There is no negative feedback. For example, a user that did not read a certain show might have done so because she dislikes the show or just because she did not know about the show or was not available to watch it. Recommender systems with such input should not concentrate only on the gathered feedback because it means considering only the positive feedback, greatly misrepresenting the full user profile. Hence, it is crucial to address also the missing data, which is where most negative feedback is expected to be found.
2. Implicit feedback is inherently noisy. Since user's behaviors are passively tracked, it is only possible to guess their preferences and true motives. For example, purchase of an item does not necessarily indicate a positive view of it. The product may have been purchased as a gift, or perhaps the user was disappointed with it.
3. The numerical value of explicit feedback indicates preference, whereas the numerical value of implicit feedback describes the frequency of actions: how much time the user watched a certain show, how frequently a user is buying a certain item, among others. That is, a higher value does not indicate a higher preference as it does with explicit feedback data. For example, the most loved show may be a movie that the user will watch only once, while there is a series that the user quite likes and thus is watching every week. However, the numerical value of the feedback is definitely useful, as it tells us about the confidence that we have in a certain observation. A one-time event might be caused by

various reasons that have nothing to do with user preferences. However, a recurring event is more likely to reflect the user opinion.

4. Evaluation of implicit-feedback recommender requires appropriate measures. In the traditional setting where a user is specifying a numeric score, there are clear metrics such as mean squared error to measure success in prediction. However with implicit models there is the need to take into account availability of the item, competition for the item with other items, and repeat feedback. For example, if we gather data on television viewing, it is unclear how to evaluate a show that has been watched more than once, or how to compare two shows that are on at the same time, and hence cannot both be watched by the user.

These characteristics prevent the direct use of algorithms that were designed with explicit feedback in mind, because they work with the relatively few known ratings while ignoring the missing ones. However, with implicit feedback it would be natural to assign values to all variables: if no action was observed then the rating r_{ui} is set to zero, meaning zero watching time, or zero purchases on record.

However, taking all user-item values as an input to the model raises serious scalability issues since the number of all those pairs tends to significantly exceed the input size since a typical user would provide feedback only on a small fraction of the available items. So, instead of modelling the matrix ratings, now a dense matrix, it is proposed a transformation to binary values:

$$p_{ui} = \begin{cases} 1 & r_{ui} > 0 \\ 0 & r_{ui} = 0 \end{cases} \quad (4)$$

That is, if a user u has consumed with the item ($r_{ui} > 0$), then it is considered as an indication that u likes i ($p_{ui} = 1$). On the other hand, if u never consumed i, no preference is assumed ($p_{ui} = 0$).

However, as discussed before, zero values of p_{ui} are associated with low confidence, because not taking any positive action on an item can derive from many other reasons beyond not liking it, for example when the user is unable to consume an item due to its price or limited availability. In addition, consuming an item also can be the result of factors different then liking it, for example when a consumer buys an item as a gift to another person. But in general, bigger r_{ui} values, give more the evidence that the user indeed likes the item. To model this measure of confidence, a set of variables c_{ui} are introduced as follows:

$$c_{ui} = 1 + \alpha r_{ui} \quad (5)$$

Where α is the rate of increased confidence that is experimentally set to α = 40.

The goal is to find a vector $x_u \in \mathbb{R}^k$ for each user u, and a vector $y_i \in \mathbb{R}^k$ for each item i that will factor user preferences. In other words, preferences are assumed to be the inner products: $p_{ui} = y_i^T x_u$. This is similar to matrix factorization techniques for explicit feedback data, described in chapter 2, with two important distinctions: we need to account for the varying confidence levels, and optimization should account for all possible u, i pairs, rather than only those corresponding to observed data. Similarly to eq. (2), factors are computed by minimizing the following cost function:

$$\min_{x,y} \sum_{(u,i)\in\kappa} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda(\|x_u\|^2 + \|y_i\|^2) \quad (6)$$

Notice that the cost function contains $m \cdot n$ terms, where m is the number of users and n is the number of items. For typical datasets $m \cdot n$ can easily reach a few billion, preventing the use of most direct optimization techniques such as stochastic gradient descent, which was widely used for explicit feedback datasets.

Alternating least squares (ALS) is proposed as an alternative efficient optimization process because when either x_u or y_i are fixed, the cost function becomes quadratic and its global minimum can be readily computed. ALS was already used for explicit feedback datasets, where unknown values were treated as missing, leading to a sparse objective function [34]. The implicit feedback setup requires a different strategy to overcome the dense cost function and to integrate the confidence levels, as follows.

The first step is recomputing all user factors x_u: assume that all x_u are gathered within an $n \times f$ matrix Y. Before looping through all users, we compute the $f \times f$ matrix $Y^T Y$ in time $O(f^2 n)$. For each user u, define the diagonal $n \times n$ matrix C^u where $C^u_{ii} = c_{ui}$, and also the vector $p(u) \in \mathbb{R}^n$ that contains all the preferences by u (the p_{ui} values). By differenciation we find an analytic expression for x_u that minimizes the cost function (6):

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u) \quad (7)$$

A computational bottleneck here is computing $Y^T C^u Y$, whose naïve calculation will require time $O(f^2 n)$ for each of the m users. A significant speedup is achieved by using the fact that $Y^T C^u Y = Y^T Y + Y^T (C^u - I) Y$. Now, $Y^T Y$ is independent of u and was already precomputed. As for the $Y^T (C^u - I) Y$, notice that $C^u - I$ has only n_u non-zero elements, where n_u is the number of items for which $r_{ui} > 0$ and typically $n_u \ll n$. Similarly, $C^u p(u)$ contains just n_u non-zero elements. Consequently, recomputation of x_u is performed in time $O(f^2 n_u + f^3)$. Here, it is assumed $O(f^3)$ time for the computation of matrix inversion$(Y^T C^u Y + \lambda I)^{-1}$. This step is performed over each of the m users, so the total running time is $O(f^2 N + f^3 m)$, where N is the overall number of non-zero observations, that is, $N = \sum_n n_u$. Importantly, running time is linear in the size of the input. Typical values of κ found experimentally are between 20 and 200.

A recomputation of the user factors is followed by a recomputation of all item factors y_i in a parallel fashion. We arrange all user-factors within an $m \times f$ matrix X. First we compute the $f \times f$ matrix $X^T X$ in time $O(f^2 m)$. For each item i, define the diagonal $m \times m$ matrix C^i where $C^i_{uu} = c_{ui}$, and also the vector $p(i) \in \mathbb{R}^m$ that contains all the preferences fot i. Then, we solve:

$$y_i = \left(X^T C^i X + \lambda I\right)^{-1} X^T C^i p(i) \quad (8)$$

Using the same technique as with the user factors, running time of this step would be $O(f^2 N + f^3 n)$. It is employed a number of sweeps of paired recomputation of user and item factors, typically 10, until they stabilize.

We highlight that the whole process scales linearly with the size of the data, while addressing the full scope of user-item pairs without resorting to any sub-sampling.

An interesting feature of the algorithm is that it allows explaining the recommendations to the end user, which is a rarity among latent factor models. This is achieved by showing a surprising and hopefully insightful link into the well-known item-oriented neighborhood approach: by replacing the user factors in Eq. (6), the predicted preference of user u for item i becomes:

$$\hat{p}_{\mathrm{ui}} = \mathrm{y_i^T x}_u = \mathrm{y_i^T}(Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u)$$

Let us denote the $\kappa \times \kappa$ matrix $(Y^T C^u Y + \lambda I)^{-1} = W^u$, which should be considered as a weighting matrix associated with user u. Accordingly, the weighted similarity between items i and j from u's viewpoint be denoted by $s_{ij} = y_i^T W^T y_j$.

Using this new notation, $\hat{p}_{ui} = \sum_{j:\, r_{uj}>0} s^u_{ij} c_{uj}$, reducing the proposed latent factor model into a linear model that predicts preferences as a linear function of past actions ($r_{uj} > 0$), weighted by item-item similarity. Each past action receives a separate term in forming the predicted $\hat{p}_{ui}$ and thus it is possible to isolate its unique contribution. The actions associated with the highest contribution are identified as the major explanation behind the recommendation. It is also possible to break the contribution of each individual past action into two separate sources: the significance of the relation to user: $u - c_{uj}$, and the similarity to target item i: $i - s^u_{ij}$.

Both Apache Mahout and MLlib libraries provide implementations of ALS for implicit feedback datasets as just described.

Chapter 5 Comparative Study

In this chapter 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.

We will first analyze how the algorithm's parameters: dimensionality of the latent factor space (k) and the regularization parameter (λ) affect the quality of the generated recommendations on both chosen domains. Then, we will show the performance gain achieved by the MLlib Spark implementation, measuring the walltime of computing the matrix factorization and also the accuracy of the generated recommendations when training the model with the best combination of parameters found in the previously analysis. Subsequently, we will study the effects on the runtime speedup if we add more machines to the cluster, as well as the scaling behavior with the two different sized datasets.

5.1 Mahout ALS Implementation

Mahout 0.9, 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 [81].

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 4.2, the parallel broadcast-join [32]. First 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 output matrices are given in sequence file format, not in RDD nor DFS format. The implementation includes the following parameters:

- `input`: directory containing files of the users' preferences that represent the ratings matrix. The file is organized as each line is a tab-delimited string, the first field is user id, which must be numeric, the second field is item id, which must be numeric and the third field is preference, which should also be a number;

- *output*: output path of the user-to-feature matrix and feature-to-item matrix;
- *lambda*: regularization parameter to avoid overfitting;
- *implicitFeedback*: boolean flag to indicate whether the input dataset contains implicit feedback;
- *alpha*: confidence parameter only used on implicit feedback;
- *numFeatures*: dimensions of feature space (number of latent factors to consider);
- *numThreadsPerSolver*: number of threads per solver mapper for concurrent execution (number of cores that you want to use per machine).;
- *numIterations*: number of iterations;
- *usesLongIDs*: boolean flag to indicate whether the input contains long IDs that need to be translated.

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

- *input*: directory containing files of user ids;
- *output*: output path of the recommended items for each input user id;
- *userFeatures*: path to the user feature matrix;
- *itemFeatures*: path to the item feature matrix;
- *numRecommendations*: maximum number of recommendations per user, default is 10;
- *maxRating*: maximum rating available;
- *numThreads*: number of threads per mapper;
- *usesLongIDs*: boolean flag to indicate whether the input contains long IDs that need to be translated;
- *userIDIndex*: index for user long IDs (necessary if usesLongIDs is true);
- *itemIDIndex*: index for item long IDs (necessary if usesLongIDs is true).

5.2 MLlib 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. The Spark implementation is shown below. Note that collection 0 until u are parallelized and collected to update each array [56]:

```
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()
}
```

The ALS recommender accepts as input an RDD of ratings (user: Int, product: Int, rating: Double). The implementation in MLlib has the following parameters [39]:

- *numBlocks*: number of blocks used to parallelize computation (set to -1 to auto-configure).
- *rank:* number of latent factors in the model.
- *iterations:* number of iterations to run.
- *lambda*: the regularization parameter in ALS.
- *implicitPrefs*: booleanflat to indicate whether to use the explicit feedback ALS variant or one adapted for implicit feedback data.
- *alpha*: parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations.

5.3 Datasets

The datasets chosen to run the experiments are from the movies domain (MovieLens) and jokes domain (Jester).

5.3.1 MovieLens

Due to copyright, Netflix data is not available for download. So, to perform the recommendation evaluation on the movies domain, the MovieLens [82] data is used. The MovieLens dataset [83] consists of anonymous ratings of movies collected by the GroupLens Research that currently operates a movie recommender based on collaborative filtering [84]. The MovieLens data set 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 data set was collected and made available by GroupLens Research at their webpage [85].

This dataset was used to evaluate matrix factorization based methods with neighbor based correction technique, and achieved a best RMSE score of 0.8275 [27].

5.3.2 Jester

The Jester dataset [86] consists of anonymous ratings of jokes collected between November 2006 and May 2009. Thus, this data is in a humor domain and it was firstly used to test the Eigentaste recommender [87] and now, it 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, and the remaining ones 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 used to evaluate matrix factorization based methods with neighbor based correction technique, and achieved a best RMSE score of 4.1229 [27].

5.4 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) [88].

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.

The Hadoop cluster configuration is done by modifying the following files:

- hadoop-env.sh: This file contains some environment variable settings used by Hadoop. You can use these to affect some aspects of Hadoop daemon behavior, such as where log files are stored, the maximum amount of heap used etc. The only variable changed was JAVA_HOME, which specifies the path to the Java 1.7.x installation used by Hadoop.
- core-site.xml: This file contains configuration settings for Hadoop Core (I/O) that are common to HDFS and MapReduce Default file system configuration. Key property – *fs.default.name* – for namenode configuration (*e.g.* hdfs://namenode/). Secondary property – *hadoop.tmp.dir* – A base for other temporary directories. It's important to note that every node needs Hadoop tmp directory.
- hdfs-site.xml: This file contains the configuration for HDFS daemons, the namenode, and datanodes. Key property – *dfs.replication* – Default block replication is 3. The default is used if replication is not specified in create time. Since we have 2 slave datanodes, this value is set to 2. Secondary property – *dfs.permissions.enabled* – with value false.
- mapred-site.xml: This file contains the configuration settings for MapReduce daemons; the job tracker and the task-trackers. Key property – *mapred.job.tracker* – hostname (or IP address) and port pair on which the Job Tracker listens for RPC communication. This parameter specifies the location of the Job Tracker for Task Trackers and MapReduce clients (*e.g.* jobtracker:8021).

5.4.1 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 the one with smaller achieved root-mean-square error (RMSE) on the validation set after running 20 iterations of the algorithm is chosen as the best fit ALS model [34]. Finally, this model is evaluated on the test set.

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

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 λ until the optimal value of 0.5. Then, the increment worsens the recommendation accuracy. To provide a better visualization only a few values of k are plotted on Figure 5-1.

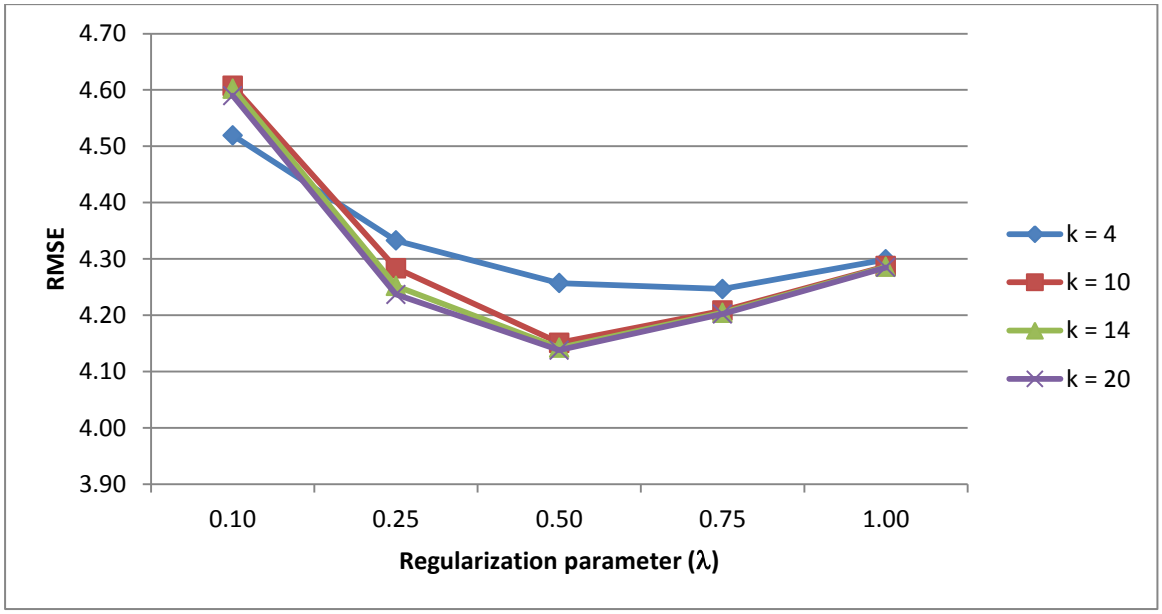


Figure 5-1: Performance of Spark-MLlib ALS with fixed k and varying λ on the Jester dataset.

Table 5-1 presents the influence of the two training parameters for Spark-MLlib ALS on the Jester validation set. The optimized parameters are $k = 20,\ \lambda = 0.5$.

k \ λ	**0.10**	**0.25**	**0.50**	**0.75**	**1.00**
2	4.2882	4.2364	4.3677	4.2413	4.2943
4	4.5193	4.3325	4.2569	4.2466	4.2990
6	4.6407	4.3473	4.2304	4.2342	4.2922
8	4.6383	4.3205	4.1833	4.2178	4.2859
10	4.6074	4.2834	4.1511	4.2080	4.2872
12	4.6200	4.2535	4.1493	4.2049	4.2857
14	4.6024	4.2520	4.1423	4.2043	4.2861
16	4.6143	4.2493	4.1390	4.2020	4.2884
18	4.5936	4.2479	4.1379	4.2026	4.2857
20	4.5898	4.2368	**4.1378**	4.2020	4.2848

Table 5-1: RMSE of Spark-MLlib ALS on Jester validation set for various k and λ.

For the HadoopMR-Mahout ALS on the Jester validation set, on Figure 5-2, we also see the same behavior presented by the Spark-MLlib implementation: the recommendation quality usually improves when increasing λ until the optimal value of 0.5 but beyond that, the recommendation is worse.

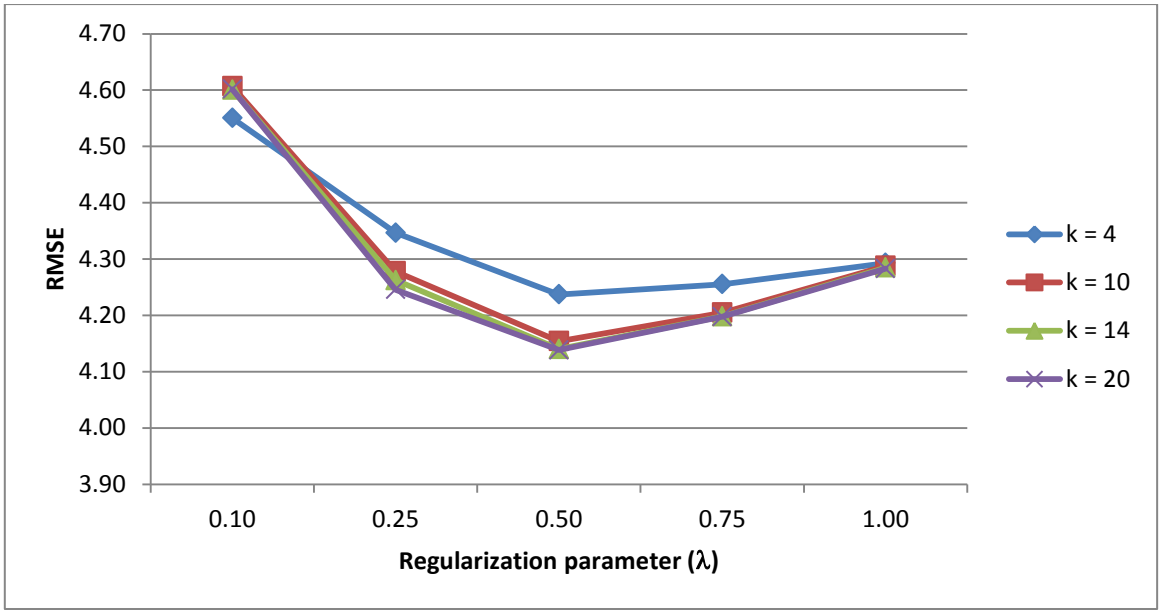


Figure 5-2: Performance of HadoopMR-Mahout ALS with fixed k and varying λ on the Jester dataset.

Table 5-2 presents the influence of the training parameters for HadoopMR-Mahout ALS on the Jester validation set. The best result found has the same parameters as the Spark-MLlib result: $k = 20,\ \lambda = 0.5$.

λ / k	**0.10**	**0.25**	**0.50**	**0.75**	**1.00**
2	4.2856	4.2373	4.2192	4.2417	4.2940
4	4.5504	4.3466	4.2371	4.2554	4.2927
6	4.6465	4.3845	4.2362	4.2252	4.2934
8	4.6326	4.3230	4.1941	4.2186	4.2863
10	4.6068	4.2779	4.1540	4.2048	4.2878
12	4.6085	4.2579	4.1446	4.2003	4.2864
14	4.6005	4.2622	4.1400	4.1982	4.2849
16	4.5989	4.2537	4.1396	4.1982	4.2846
18	4.5983	4.2511	4.1359	4.1969	4.2855
20	4.6014	4.2457	**4.1385**	4.1976	4.2830

Table 5-2: RMSE of HadoopMR-Mahout ALS on Jester validation set for various k and λ.

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 λ, but the MLlib 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 5-3.

	Spark-MLlib	**HadoopMR-Mahout**
$k_{Best\ fit\ ALS}$	20	20
$\lambda_{Best\ fit\ ALS}$	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

Table 5-3: Best fit ALS model results for the Jester dataset

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, presented on Table 5-4. We can see 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.

λ / k	**0.10**	**0.25**	**0.50**	**0.75**	**1.00**
2	0.8430	0.9001	1.0034	1.1490	1.3258
4	0.8267	0.8920	1.0034	1.1490	1.3258
6	0.8192	0.8977	1.0034	1.1490	1.3258
8	0.8146	0.8944	1.0034	1.1490	1.3258
10	0.8127	0.8939	1.0034	1.1490	1.3258
12	0.8113	0.8938	1.0034	1.1490	1.3258
14	0.8109	0.8937	1.0034	1.1490	1.3258
16	0.8104	0.8906	1.0034	1.1490	1.3258
18	0.8101	0.8936	1.0034	1.1490	1.3258
20	**0.8099**	0.8929	1.0033	1.1490	1.3258

Table 5-4: RMSE of Spark-MLlib ALS on MovieLens validation set for various k and λ.

The HadoopMR-Mahout ALS modelling results are detailed on Table 5-5. 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.

k \ λ	**0.10**	**0.25**	**0.50**	**0.75**	**1.00**
2	0.8509	0.8960	1.0036	1.1489	1.3260
4	0.8344	0.8957	1.0036	1.1489	1.3260
6	0.8276	0.8957	1.0036	1.1489	1.3260
8	0.8227	0.8958	1.0036	1.1489	1.3260
10	0.8218	0.8958	1.0036	1.1489	1.3260
12	0.8209	0.8959	1.0036	1.1489	1.3260
14	0.8203	0.8959	1.0036	1.1489	1.3260
16	0.8198	0.8960	1.0036	1.1489	1.3260
18	0.8198	0.8960	1.0036	1.1489	1.3260
20	**0.8196**	0.8961	1.0036	1.1489	1.3260

Table 5-5: RMSE of HadoopMR-Mahout ALS on MovieLens validation set for various k and λ.

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. The results for the Jester dataset are summarized below, on Table 5-6.

	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

Table 5-6: Best fit ALS model results for the MovieLens dataset.

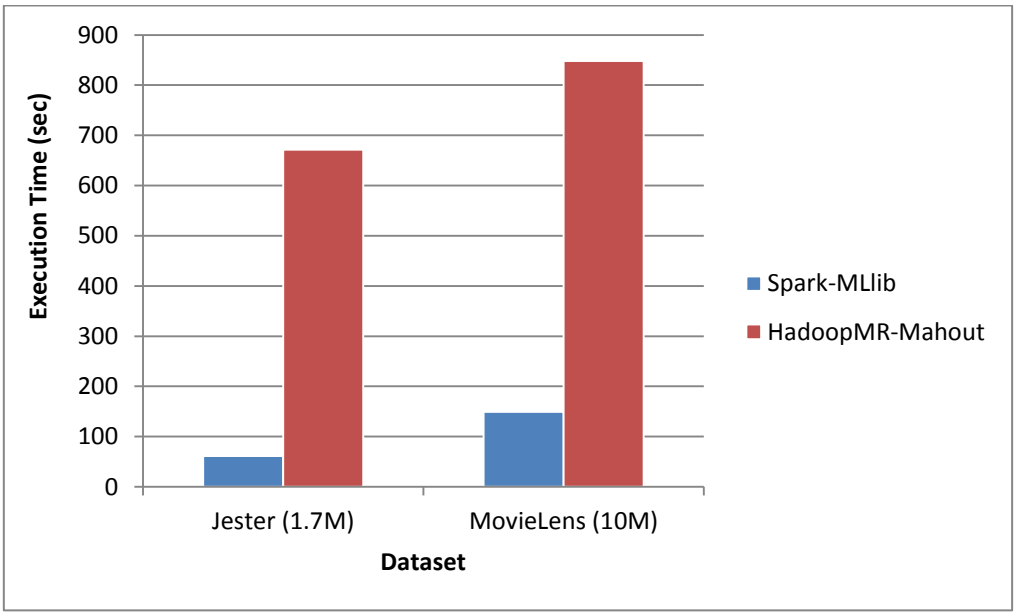


Figure 5-3: Execution time for the best fit ALS model on a cluster with 4 machines.

5.4.2 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.

By comparing the results on the two datasets, MovieLens which has 5.9 times more ratings than the Jester dataset, the execution time difference for the Spark-MLlib implementation show that the biggest gap is when working on a cluster with only two machines, when training the model for the MovieLens dataset takes 174 seconds more than training for the Jester dataset (Figure 5-4). This means that the MovieLens dataset training time is 170% slower comparing to the Jester dataset. Then, with 4 and 6 machines the gap decreases and on average the bigger dataset takes 140% more time to be trained.

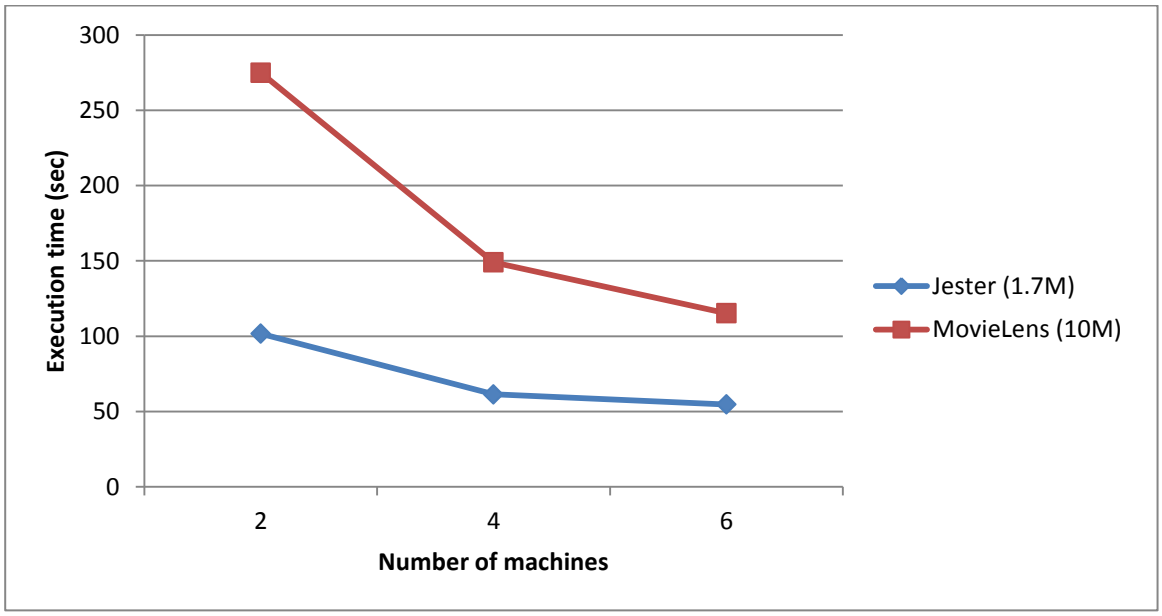


Figure 5-4: Execution time of Spark-MLlib ALS implementation for a growing number of machines.

The execution time for the HadoopMR-Mahout implementation, found on Figure 5-5, show that the difference to train the models on the two datasets is smaller than it is on the Spark-MLlib implementation. The biggest difference is found when running a four machine cluster, but in this case it is only 26% slower to train the MovieLens then the Jester dataset. In average, when running the HadoopMR-Mahout ALS, it takes 22% more time to train the bigger dataset, or about 156 seconds more.

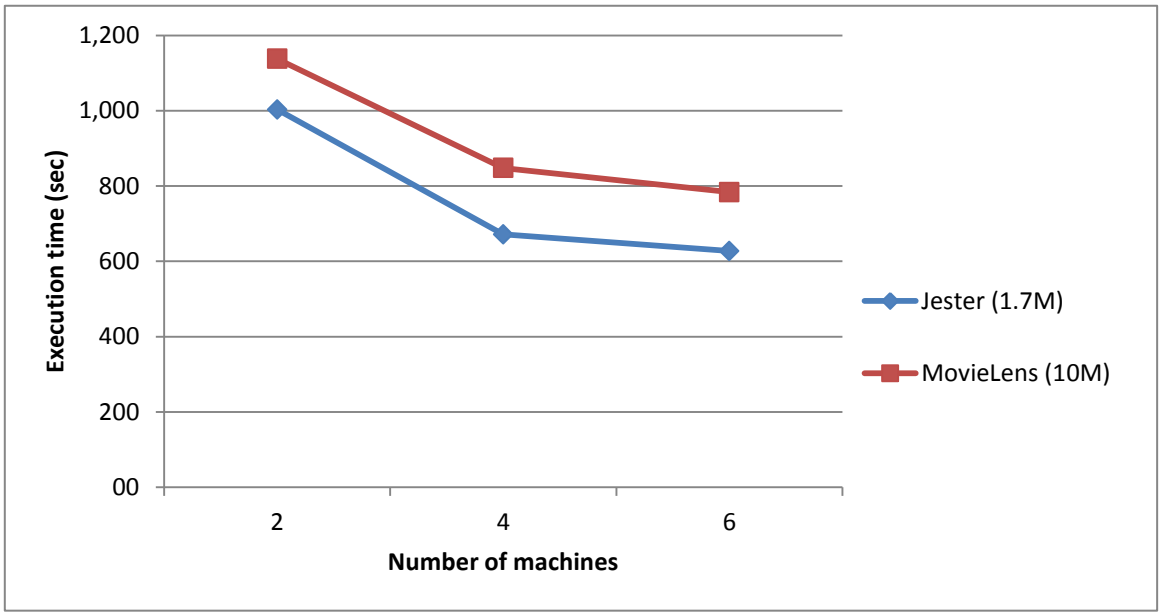


Figure 5-5: Execution time of HadoopMR-Mahout ALS implementation for a growing number of machines.

Comparing the speedup values for the two implementations, shown on Figure 5-6, 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.

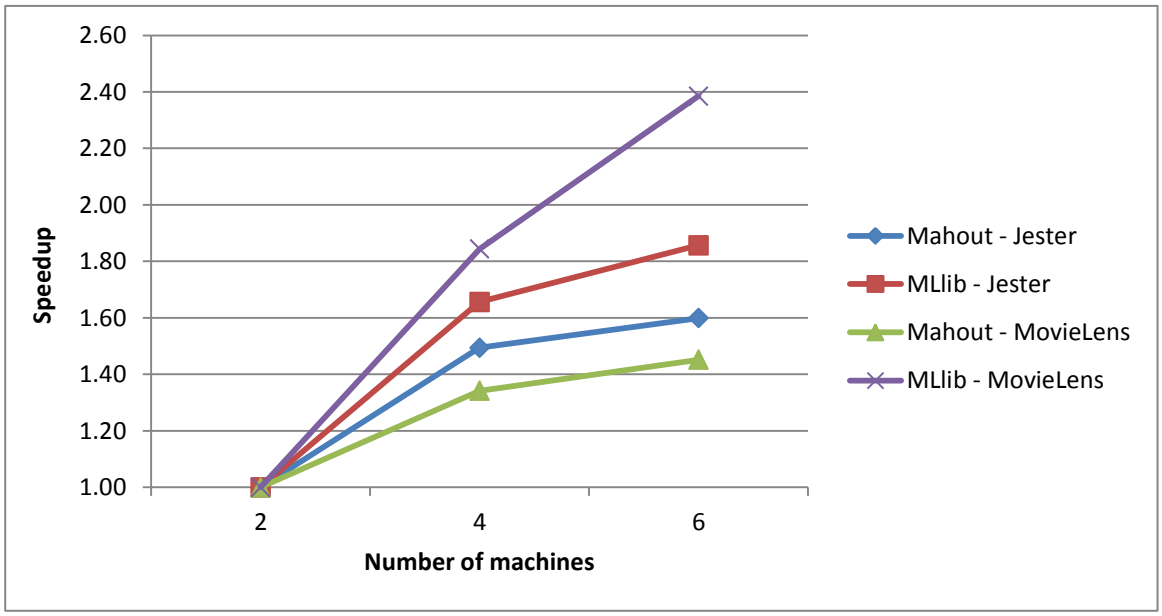


Figure 5-6: Speedup for both implementations on Jester and MovieLens datasets

The distributed and parallel ALS implementation on MLlib 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 [88], which suggests that the Spark implementation is suitable for real world use cases.

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

Table 5-7: Summary of Recommendation time

5.5 Discussion

In this chapter we evaluated two different parallel implementations of the ALS algorithm in two different datasets. The experiments compare accuracy of the obtained recommendations, the

modeling execution time and the scalability of the solutions. These topics represent the main concerns when creating a recommender system for an online application nowadays, since with the growth of available content and internet usage traditional single-machine or single-threaded computing is no longer viable. Parallel and/or distributed computing becomes an essential component for any computing environment.

By executing each implementation on two different sized datasets, from different recommendation domains, we demonstrate that the Spark framework has a significative performance improvement when compared to the Hadoop MapReduce framework solution, thus being a more appropriate choice. The results on the recommendation problem show that ALS implemented on MLlib has a faster modeling time and the items recommended have better quality than the Mahout implementation.

Also, the Spark-MLlib implementation has a better speedup as well: when training the models on a cluster with 2 machines it had an execution time 86% faster on the Jester dataset and 139% on the MovieLens dataset, while only 60% faster for the Jester dataset and 45% for the MovieLens dataset on the HadoopMR-Mahout implementation.

We would like to remark that MLlib can also scale to much larger datasets and to larger number of nodes, thanks to its fault-tolerance design, which makes it a solution for implementing online recommender systems on websites with high traffic and/or big database of users.

In the next chapter, we discuss the contributions and limitations of the comparative study, also presenting topics for future research.

Chapter 6 Conclusion

We presented a review with a focus on recommender systems theory, focusing on the class of collaborative filtering techniques that are fast and easy to calculate, called latent factor models. This kind of technique addresses challenges faced mainly by online recommender systems, which process real-time information to build predictive models and present the generated recommendations in seconds. Performance problems become more evident when trying to construct recommender systems associated with websites that have a large number of users and items associated with huge databases. An efficient, distributed matrix factorization of large and sparse matrices that runs on clusters of commodity machines is crucial to applying latent factor models in such industrial-scale recommender systems.

We explored the Alternating Least Squares (ALS) algorithm as an efficient approach in situations where generating online recommendations and processing large datasets is required. This is because ALS steps can be calculated independently, allowing massive parallelization on the algorithm's implementation. We described two scalable and data-parallel implementations of the ALS algorithm, the Mahout ALS and MLlib ALS. Each one uses a different framework for distributed processing on clusters of commodity hardware: Hadoop MapReduce and Spark respectively.

Then, we propose a methodology for comparing the different implementations of the ALS algorithm for collaborative filtering recommender systems, on 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, that is, we trained multiple ALS models with a combination of different parameter values and found the combination that had lower RMSE on the validation set. Using the optimized parameters to train the ALS models we present an evaluation of the implementations in terms of execution time and accuracy results on the test set.

The experimental results show that, by comparing the best fit ALS models achieved by both implementations, Spark-MLlib solution has a better performance than the Mahout ALS in terms of accuracy and efficiency for the two recommendation domains testes. For the Jester dataset the RMSE on the test set with Spark-MLlib is 0.13% better than with HadoopMR-Mahout, and the training time is more than 10 times faster in a cluster with 4 machines. For the MovieLens dataset the RMSE on the test set is 1.4% smaller on the Spark-MLlib implementation, and the modeling was 5 times faster to run.

We empirically show that ALS implemented on MLlib and executed on the Spark framework is more efficient in both tested cases. Mainly because it features an advanced Directed Acyclic Graph (DAG) execution engine that supports cyclic data flow and in-memory computing that helps to

eliminate the Hadoop MapReduce multi-stage execution model. This offers significant performance improvements for the recommendation problem, not just in efficiency but also in terms of accuracy of the recommendation, which translates into online display of relevant and personalized content, a relevant issue when deploying a recommender system on the web.

This study also features a scalability experiment, running the best fit ALS model on a cluster of 2, 4 and 6 machines. We found the results very favorable to Spark again, 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 takes 115.3 seconds for a dataset containing about 10 million ratings, and by extrapolation it would take about 415 seconds 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 measured in a few seconds, which is a suitable timeframe for production settings.

6.1 Future Work

Future works are desirable in order to keep comparing the recommendation algorithms implementations available in the newer releases of MLlib and Mahout, since both engines for large-scale data processing are rapidly evolving.

Spark is now at version 1.4.1, released on July 2015. Since release 1.3 a new DataFrames API was introduced, that provides powerful and convenient operators when working with structured datasets. The DataFrame is an evolution of the base RDD API that includes named fields along with schema information. Also, MLlib has received updates on the ALS implementation that lead to significant performance gain. Since release 1.4 they provide the SparkR, an R binding for Spark based on Spark’s new DataFrame API. SparkR gives R users access to Spark’s scale-out parallel runtime along with all of Spark’s input and output formats. It also supports calling directly into Spark SQL.

On the other hand, Mahout 0.10 released in April 2015 was a major release, which separates out a machine learning environment called Mahout-Samsara that runs fully on Spark. The Hadoop MapReduce versions of Mahout algorithms are still maintained but no new MapReduce contributions are accepted and Mahout Samsara based algorithms include a Distributed regularized Alternating Least Squares (dals) [89]. This change to Spark as an execution engine, which will hopefully address the performance concerns of the version of Mahout evaluated in this work, which runs on Hadoop

MapReduce and thus suffers from the scheduling overhead and lack of support for iterative computation that substantially slows down ALS.

Finally, a direct way to improve our work is by validation the algorithms on larger datasets, with millions or billions of ratings, and if possible from different recommendation domains such as music, products and news.

Bibliography

[1] L. R. a. B. S. Francesco Ricci, “Introduction to Recommender Systems Handbook,” em *Recommender Systems Handbook*, Springer, 2011, pp. 1-35.

[2] Netflix, “Netflix,” Netflix, 2015. [Online]. Available: http://www.netflix.com/browse. [Last checked June 2015].

[3] Netflix Prize, “Netflix Prize,” 08 2009. [Online]. Available: http://www.netflixprize.com//community/viewtopic.php?id=1537. [Last checked June 2015].

[4] P. M. a. V. Sindhwani, “Recommender Systems,” *Encyclopedia of Machine Learning*, 2010.

[5] V. C. F. F. Fidel Cacheda, “Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems,” *ACM Transactions on the Web (TWEB)*, vol. 5, n. 1, February 2011.

[6] a. G. K. Christian Desrosiers, “A Comprehensive Survey of Neighborhood-based Recommendation Methods,” em *Recommender Systems Handbook*, Springer, 2011, pp. 107-144.

[7] G. K. ,. J. K. ,. J. R. Badrul Sarwar, “Item-based collaborative filtering recommendation algorithms,” em *Proceedings of the 10th International Conference on World Wide Web*, 2001.

[8] Spotify, “Spotify,” [Online]. Available: https://www.spotify.com/br/. [Last checked June 2015].

[9] H. C. ,. D. Z. Zan Huang, “Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering,” *ACM Transactions on Information Systems*, pp. 116-142, January 2004.

[10] D. H. ,. C. K. John S. Breese, “Empirical Analysis of Predictive Algorithms for Collaborative Filtering,” em *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, 1998.

[11] A. M. ,. G. H. Ruslan Salakhutdinov, “Restricted Boltzmann Machines for Collaborative Filtering,” em *Proceedings of the 24 th International Conference on Machine Learning*, 2007.

[12] G. K. J. A. K. J. T. R. Badrul M. Sarwar, “Application of Dimensionality Reduction in Recommender System—A Case Study,” em *ACM WEBKDD WORKSHOP*, 2000.

[13] Y. Koren, “Factorization meets the neighborhood: a multifaceted collaborative filtering model,” em *Proceedings of the 14th ACM SIGKDD conference*, 2008.

[14] “Investigation of various matrix factorization methods for large scale recommender systems,” em *IEEE International Conference on Data Mining Workshops, 2008. ICDMW '08*, 2008.

[15] V. C. D. F. V. F. Fidel Cacheda, “Comparison of Collaborative Filtering Algorithms: Limitations of Current Techniques and Proposals for Scalable, High-Performance Recommender Systems,” *ACM Transactions on the Web (TWEB),* February 2001.

[16] C. Johnson, “Scala Data Pipelines for Music Recommendations,” 2015. [Online]. Available: http://www.slideshare.net/MrChrisJohnson/scala-data-pipelines-for-music-recommendations [Last checked June 2015].

[17] Globo.com, “Globo.com,” [Online]. Available: http://www.globo.com/. [Last checked June 2015].

[18] Globo.com, “Por que trabalhar na globo.com?,” [Online]. Available: https://github.com/globocom/IwantToWorkAtGloboCom. [Last checked June 2015].

[19] Amazon.com, “Amazon.com,” [Online]. Available: http://www.amazon.com/. [Last checked June 2015].

[20] ExportX, “How Many (More) Products Does Amazon Sell?,” August 2014. [Online]. Available: http://export-x.com/2014/08/14/many-products-amazon-sell-2. [Last checked June 2015].

[21] Statista, “Number of worldwide active Amazon customer accounts from 1997 to 2014 (in millions),” 2014. [Online]. Available: http://www.statista.com/statistics/237810/number-of-active-amazon-customer-accounts-worldwide/. [Last checked June 2015].

[22] Twitter, “Twitter,” [Online]. Available: www.twitter.com. [Last checked June 2015].

[23] Statista, “Number of monthly active Twitter users worldwide from 1st quarter 2010 to 1st quarter 2015 (in millions),” 2015. [Online]. Available: http://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/. [Last checked June m 2015].

[24] Facebook, “Facebook,” [Online]. Available: www.facebook.com. [Last checked June 2015].

[25] Statista, “Number of monthly active Facebook users worldwide as of 1st quarter 2015 (in millions),” 2015. [Online]. Available: http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/. [Last checked June 2015].

[26] Y. K. a. R. Bell, “Advances in Collaborative Filtering,” em *Recommender Systems Handbook*, Springer, 2011, pp. 145-186.

[27] I. P. B. N. D. T. Gábor Takács, “Scalable Collaborative Filtering Approaches for Large Recommender Systems,” *Journal of Machine Learning Research,* vol. 10, pp. 623-656, January 2009.

[28] R. B. a. C. V. Yehuda Koren, “Matrix Factorization Techniques for Recommender Systems,” *Computer,* vol. 42, pp. 30-37, August 2009.

[29] Y. K. a. C. V. Robert M. Bell, “Modeling Relationships at Multiple Scales to Improve Accuracy of Large Recommender Systems,” em *Proeedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2007.

[30] A. Paterek, “Improving regularized singular value decomposition for collaborative filtering,” em *Proc. KDD Cup and Workshop*, 2007.

[31] I. P. B. N. D. T. Gabor Takács, “Major components of the Gravity Recommendation System,” *SIGKDD Explorations 9*, pp. 80-84, 2007.

[32] C. B. M. S. A. A. M. Sebastian Schelter, “Distributed Matrix Factorization with MapReduce using a series of Broadcast-Joins,” em *Recommender Systems*, 2013.

[33] The Apache Software Foundation, “Apache Hadoop,” [Online]. Available: http://hadoop.apache.org/. [Last checked June 2015].

[34] D. W. R. S. a. R. P. Yunhong Zhou, “Large-scale Parallel Collaborative Filtering for the Netflix Prize,” em *Proc. 4th Int’l Conf. Algorithmic Aspects in Information and Management, LNCS* , 2008.

[35] The Apache Software Foundatio, “MLlib,” [Online]. Available: http://spark.apache.org/mllib/. [Last checked June 2015].

[36] Y. K. ,. C. V. Yifan Hu, “Collaborative Filtering for Implicit Feedback Datasets,” em *Proc. IEEE Int’l Conf. Data Mining (ICDM 08)*, 2008.

[37] J. Lin, “MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That’s Not a Nail!,” *CoRR*, 2012.

[38] The Apache Software Foundation, “Apache Spark,” [Online]. Available: http://spark.apache.org/. [Last checked June 2015].

[39] The Apache Software Foundation, “MLlib - Collaborative Filtering,” [Online]. Available: https://spark.apache.org/docs/1.1.0/mllib-collaborative-filtering.html. [Last checked June 2015].

[40] B.-J. Y. Dohyun Kim, “Collaborative Filtering Based on Iterative Principal Component Analysis,” *Expert Systems with Applications*, vol. 38, n. 4, p. 823–830, May 2005.

[41] Y. K. a. C. V. Robert M. Bell, “Modeling Relationships at Multiple Scales to Improve,” em *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, New York, 2007.

[42] S. Funk, “Netflix Update: Try This at Home,” December 2006. [Online]. Available: http://sifter.org/~simon/journal/20061211.html. [Last checked June 2015].

[43] “Improving Regularized Singular Value Decomposition for Collaborative Filtering,” em *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and*

data mining, New York, NY, 2007.

[44] A. M. Ruslan Salakhutdinov, "Probabilistic Matrix Factorization," *Advances in Neural Information Processing Systems*, pp. 1257-1264, 2008.

[45] A. N. Tikhonov, "Solution of incorrectly formulated problems and the regularization method," *Soviet Mathematics*, vol. 4, p. 1035–1038, 1963.

[46] R. B. ,. C. V. Yehuda Koren, "Matrix Factorization Techniques for Recommender Systems," *Computer*, vol. 42, pp. 30-37 , 2009.

[47] P. J. H. E. N. Y. S. Rainer Gemulla, "Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent," em *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011.

[48] The Apache Software Foundation. , "Apache Hadoop," 2014. [Online]. Available: https://hadoop.apache.org/. [Last checked June 2015].

[49] The Apache Software Foundation, "Apache Pig," [Online]. Available: http://pig.apache.org/. [Last checked June 2015].

[50] The Apache Software Foundation, "Apache Hive," [Online]. Available: http://hive.apache.org/. [Last checked June 2015].

[51] The Apache Software Foundation, "HBase," [Online]. Available: http://hbase.apache.org/. [Last checked June 2015].

[52] The Apache Software Foundation, "Apache Mahout," 2014. [Online]. Available: https://mahout.apache.org/. [Last checked June 2015].

[53] The Apache Software Foundation, "HDFS Architecture," 2015. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. [Last checked June 2015].

[54] J. D. a. S. Ghemawat, "MapReduce: simplified data processing on large clusters," em *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* , Berkeley, CA, USA, 2004.

[55] M. G. Noll, "Running Hadoop on Ubuntu Linux (Multi-Node Cluster)," 2004-2015. [Online]. Available: http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/. [Last checked June 2015].

[56] M. C. M. J. F. S. S. I. S. Matei Zaharia, "Spark: Cluster Computing with Working Sets," em *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[57] A. Grigorev, "Apache Mahout Samsara: The Quick Start," 2015. [Online]. Available: http://www.itshared.org/2015/04/apache-mahout-samsara-quick-start.html. [Last checked June

2015].

[58] The Apache Software Foundation, “Apache Mahout 0.10.0 Release Notes,” 2015. [Online]. Available: http://mahout.apache.org/release-notes/Apache-Mahout-0.10.0-Release-Notes.pdf. [Last checked June 2015].

[59] The Apache Software Foundation, “Apache Spark,” 2015. [Online]. Available: http://spark.apache.org/. [Last checked June 2015].

[60] AMPLab, “AMPLab UC Berkeley,” 2015 . [Online]. Available: https://amplab.cs.berkeley.edu/. [Last checked June 2015].

[61] The Apache Software Foundation, “Project History,” [Online]. Available: http://spark.apache.org/community.html#history. [Last checked June 2015].

[62] The Apache Software Foundation, “Spark SQL Programming Guide,” [Online]. Available: http://spark.apache.org/docs/1.1.1/sql-programming-guide.html. [Last checked June 2015].

[63] The Apache Software Foundation, “Spark Streaming Programming Guide,” [Online]. Available: http://spark.apache.org/docs/1.1.1/streaming-programming-guide.html. [Last checked June 2015].

[64] LAPACK Project Software, “LAPACK — Linear Algebra PACKage,” [Online]. Available: http://www.netlib.org/lapack/. [Last checked June 2015].

[65] The Apache Software Foundation, “GraphX Programming Guide,” [Online]. Available: http://spark.apache.org/docs/1.1.1/graphx-programming-guide.html. [Last checked June 2015].

[66] The MathWorks Inc., “Matlab,” 2015. [Online]. Available: http://www.mathworks.com/product/matlab/. [Last checked June 2015].

[67] Netflix Prize, “Netflix Prize,” [Online]. Available: http://www.netflixprize.com/. [Last checked June 2015].

[68] Netflix, “Netflix Prize Leaderboard,” 2009. [Online]. Available: http://www.netflixprize.com/leaderboard. [Last checked June 2015].

[69] BellKor's Pragmatic Chaos, “BellKor's Pragmatic Chaos is the winner of the $1 Million Netflix Prize,” 2009. [Online]. Available: http://www2.research.att.com/~volinsky/netflix/bpc.html. [Last checked June 2015].

[70] Netflix, “The BellKor Solution to the Netflix Grand Prize,” 2009. [Online]. Available: http://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf. [Last checked June 2015].

[71] E. N. P. J. H. Y. S. Rainer Gemulla, “Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent,” em *Proceedings of the 17th ACM SIGKDD international*

conference on Knowledge discovery and data mining, 2011.

[72] B. R. ,. C. R. ,. S. J. W. Feng Niu, "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," em *Neural Information Processing Systems Conference*, 2011.

[73] F. M. a. R. G. Christina Teflioudi, "Distributed Matrix Completion," em *IEEE International Conference on Data Mining*, 2012.

[74] Netflix, "Netflix Recommendations: Beyond the 5 stars (Part 1)," April 2012. [Online]. Available: http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html. [Last checked June 2015].

[75] ACM, "Workshop on Large Scale Recommendation Systems," 2013. [Online]. Available: http://recsys.acm.org/recsys13/largescale/. [Last checked June 2015].

[76] SlideShare, "Recommendation at Netflix Scale," 2013. [Online]. Available: http://pt.slideshare.net/justinbasilico/recommendation-at-netflix-scale?next_slideshow=1. [Last checked June 2015].

[77] S. Schelter, "GitHub," [Online]. Available: https://github.com/sscdotopen/recsys-als. [Last checked June 2015].

[78] J. M. P. E. J. R. E. J. S. Y. T. Spyros Blanas, "A comparison of join algorithms for log processing in mapreduce," em *Proceedings of the ACM SIGMOD International Conference on Management of data*, 2010.

[79] M. L. Braun, "JBlas - Linear Algebra for Java," [Online]. Available: http://jblas.org. [Last checked June June 2015].

[80] C. Johnson, "Github," [Online]. Available: https://github.com/MrChrisJohnson/implicit-mf. [Last checked June 2015].

[81] Apache, "Introduction to ALS Recommendations with Hadoop," Mahout, 2014. [Online]. Available: https://mahout.apache.org/users/recommender/intro-als-hadoop.html. [Last checked June 2015].

[82] GroupLens, "MovieLens," [Online]. Available: https://movielens.org/. [Last checked June 2015].

[83] GroupLens, "MovieLens dataset," [Online]. Available: http://grouplens.org/datasets/movielens/. [Last checked June 2015].

[84] GroupLens, " MovieLens Recommender," [Online]. Available: https://movielens.org. [Last checked June 2015].

[85] GroupLens, "MovieLens," 2015. [Online]. Available: http://grouplens.org/datasets/movielens/.

[Last checked June 2015].

[86] Berkeley, “Jester dataset,” [Online]. Available: http://eigentaste.berkeley.edu/dataset/. [Last checked June 2015].

[87] T. R. D. G. a. C. P. Ken Goldberg, “Eigentaste: A Constant Time Collaborative Filtering Algorithm,” *Information Retrieval,* pp. 133-151, July 2001.

[88] Amazon Web Services, “Amazon EC2 Instances,” 2015. [Online]. Available: http://aws.amazon.com/ec2/instance-types/?nc2=h_ls. [Last checked June 2015].

[89] The Apache Software Foundation, “Apache Mahout Release Notes,” Apache Mahout, [Online]. Available: http://mahout.apache.org/general/release-notes.html. [Last checked June 2015].

[90] A. P. ,. L. H. U. ,. D. M. P. Andrew I. Schein, “Methods and metrics for cold-start recommendations,” em *n Proceedings of the 25th Annual International ACM SIGIR Conference on Research*, 2002.

[91] D. W. O. a. J. Kim, “Implicit Feedback for Recommender Systems,” em *Proc. 5th DELOS Workshop on Filtering and Collaborative Filtering*, 1998.