An implementation of a Distributed Stochastic Gradient Descent for Recommender Systems based on Map-Reduce

Manuel Pozo, Raja Chiky

AN IMPLEMENTATION OF A DISTRIBUTED STOCHASTIC GRADIENT DESCENT FOR RECOMMENDER SYSTEMS BASED ON MAP-REDUCE

Manuel Pozo and Raja Chiky

Institut Supérieur d'Eléctronique de Paris
LISITE Lab - http://lisite.isep.fr
28, rue Notre-Dame-des-Champs. 75006 Paris, France.
manuel.pozo, raja.chiky@isep.fr

ABSTRACT

This work presents an implementation of a Distributed Stochastic Gradient Descent (DSGD) for Recommender Systems based on Hadoop/MapReduce. Recommender Systems aim at presenting first the information in which users may be more interested. To do this, they analyse a great volume of data that represent the users preferences (e.g. ratings). Thus, this stirs up the need of load-balancing. DSGD is a Matrix Factorization technique that has demonstrated high accuracy and scalability. In this work we expose this algorithm and modify it to improve its accuracy and adaptability to a hadoop cluster. The experimentation phase uses MovieLens datasets. Comparisons with other algorithms are given. Results show the good performance of the implementation.

***Index Terms*—** recommender system, collaborative filtering, matrix factorization, gradient descent, parallelization.

1. INTRODUCTION

The web content has become so vast that users hardly find the information they are looking for. Recommender Systems aim at selecting and presenting first the information in which users may be more interested. Nowadays, enterprises use these systems to personalize items for users, i.e. recommendations, by analysing their preferences [1]. For instance, Amazon and Youtube propose new products/videos regarding what is already bought/watched. A very interesting datum has been offered by the video media service Netflix in 2014 [2]: improving the accuracy of recommendations by 10% may lead in around $500M per month of outcomes. Hence, these systems are becoming more and more important in the past decade.

The interest of users in items can be explicit (e.g. giving a rating to a movie) or implicit (e.g. tracking navigational behaviour). Recommender Systems analyze this feedback in order to generate recommendations [1]. Furthermore, the number of users and items uses to be high. For instance, Netflix has around 20 millions customers, 80 thousands movies and 5 billions ratings [3]. Normally users just rate some of these items. This yields in a sparse dataset that contains a huge volume of missing/unknown ratings to predict. Thus, these predictions are computationally expensive. Novel recommenders should be accurate to fit users interests, and scalable to alleviate time processing. Matrix Factorization (MF) is a recommendation technique that has demonstrated great accuracy and scalability for very large datasets [4, 5].

This work focuses on a particular Matrix Factorization technique called Distributed Stochastic Gradient Descent (DSGD). The contribution in this paper are: (1) we detail a MapReduce implementation of this technique in a real Hadoop environment, (2) we did a modification in the paradigm distribution to achieve better accuracy and better adaptability to the hadoop cluster. The experimentation phase uses the publicly available MovieLens dataset. Main techniques are compared in terms of recommendation accuracy and time-consumption. The results indicate the good performance of the implementation.

This article is structured as follows: section 2 presents a brief state of the art in Matrix Factorization. In sections 3 and 4, we explain the distribution paradigm for Stochastic Gradient Descent techniques and the proposed modification. The experimentation phase is given in section 5. Finally, in section 6 conclusions and future work are exposed.

2. RELATED WORK

This state of the art mainly focuses on Matrix Factorization since it has demonstrated high accuracy predicting recommendations [4]. It relies on collaborative filtering assumptions: users who agreed in past tend to agree in future. Hence, it groups people of similar tastes, and it recommends past liked items from people with the same preferences.

The Matrix Factorization technique decomposes a matrix R into two random matrices, P and Q, in such a way that the multiplication of both matrices gives approximately the original one [5]. This concept is used in recommender systems to predict the missing ratings or unknown values of a sparse rating matrix R. Typically, known ratings in the matrix are de-

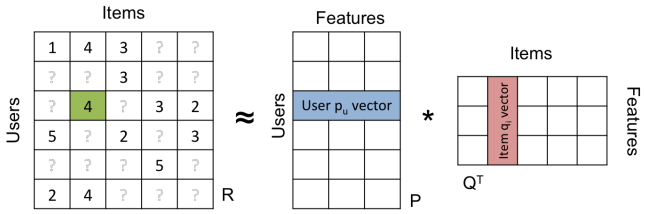

Fig. 1. Matrix Factorization technique. Decomposition in matrices P and Q.

noted by $r_{u,i}$, where u stands for a user-row and i for an item-column. Thus, $r_{u,i}$ can be obtained by the multiplication of a vector p_u from matrix P and a vector q_i from matrix Q. The dimensions of these vectors are latent factors, a.k.a. latent features, without any special semantic meaning used to represent information. Figure 1 illustrates this technique.

As a consequence, the goal is to find the matrices P and Q that best approximate known entries in R. That is, to look for the best vectors so that a prediction $\hat{r}_{u,i}$ is close to its real value $r_{u,i}$. This creates a basic baseline predictor $\hat{r}_{u,i} = p_u \cdot q_i^t$. Thus, other missing values might be discovered by minimizing the quadratic error of the difference between real and predicted values: $min \sum_{r_{u,i} \in R} (r_{u,i} - \hat{r}_{u,i})^2$.

Indeed, this problem can be solved by using optimization algorithms. The two most known optimization techniques that may find out accurate predictions are based on alternating minimization and gradient descent [6].

On the one hand, alternating minimization techniques has simple algebraic resolution. Popularized by the Alternating Least Square (ALS) [7–11], this technique decomposes the problem into two simple optimization problems represented in P and Q. Then, by fixing one matrix, the other one has to be guessed. Iterating the fixed matrix in order to guess the other one yields in an approximated result for R. In order to minimize the error, all ratings are consider in each iteration. On the other hand, the gradient descent optimization technique includes learning-parameters that study the ratings patterns to improve the results of the algorithm. This technique was popularized as Gradient Descent (GD) and Stochastic Gradient Descent (SGD) [12–14]. In order to minimize the error, it iterates among each single entry in R looking for a global minimum. After each iteration, the parameters are updated taking the negative gradient of the function into account, what improves the accuracy by making little steps per iteration. In [6, 15, 15] authors detail the evolution of this technique to exploit both explicit and implicit feedback.

SGD techniques are more accurate than ALS techniques [3, 6]. However, it is difficult to implement in a parallel way. As a consequence, there is a tendency to use ALS techniques due to their good accuracy and easy scalability [11, 16]. Yet, a highly paralellizable version of SGD has been recently proposed in [3, 16], which has demonstrated that SGD may overcome ALS in both accuracy and scalability. In this work we offer an implementation of the Distributed SGD algorithm explained in [3]. In addition, we have slightly changed this algorithm to achieve better accuracy and better adaptability to the cluster. The following sections will explain the paradigm in the distribution of SGD and our contribution.

3. DISTRIBUTED STOCHASTIC GRADIENT DESCENT (DSGD)

3.1. Stochastic Gradient Descent (SGD)

Gradient Descent (GD) is based on Matrix Factorization. It goals to explain the ratings in a matrix R by using two matrices P and Q. The baseline predictor is defined as $\hat{r}_{u,i} = \mu + b_u + b_i + p_u \cdot q_i^t$, where μ is the average of the ratings in the matrix R, and b_u, b_i are biases for the user u and the item i respectively, which represent a deviation from the average. In order to learn this parameters and to find the best P and Q, the goal is to minimize the λ-regularized squared error of:

$$min_{b_u,bi,p_u,q_i} \sum_{r_{u,i} \in R} (r_{u,i} - \hat{r}_{u,i})^2 + \lambda \cdot (b_u^2 + b_i^2 + \|p_u\|^2 + \|q_i\|^2)$$

This technique iterates among ratings one by one evaluating the error in the prediction, $e_{u,i} = r_{u,i} - \hat{r}_{u,i}$. Stochastic Gradient Descent proposes to iterate over a batch of ratings instead, what allows a faster convergence in very large datasets. Then, they update users and items parameters and matrices by taking the negative gradient, where a learning rate γ is introduced:

- $b_u := b_u + \gamma \cdot (e_{u,i} - \lambda \cdot b_u)$
- $b_i := b_i + \gamma \cdot (e_{u,i} - \lambda \cdot b_i)$
- $p_u := p_u + \gamma \cdot (e_{u,i} \cdot p_u - \lambda \cdot q_i)$
- $q_i := q_i + \gamma \cdot (e_{u,i} \cdot q_i - \lambda \cdot p_u)$

However, either simple iteration or batch iterations, one can highlight the interdependency of the new updated values and previous values [3]. For instance, new value for b_u depends on last computed value of b_u for this user u. In a single machine, new updated values after one iteration are ready for a new iteration. On the contrary, parallelizing the algorithm may lead in miss-updated values or lack of synchronization.

Avoiding this dependency is possible by analysing the matrix R. In fact, independent entries can be found, and thus, can be computed in parallel. Indeed, independent entries do not share rows and/or columns (users and/or items). As a consequence, they have no common parameters; hence, there is no dependency. Then, computing only independent entries at a time may allow to safely update parameters for future iterations. Based on this idea, authors in [3, 16] have developed a fully distributed stochastic gradient descent, which is described below.

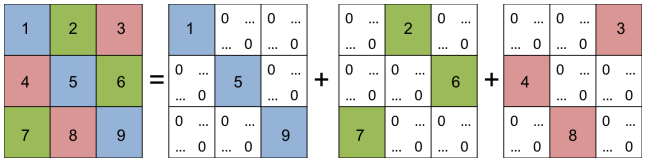

Fig. 2. Stratification and Block decomposition. Blocks 1,5, 9 do not share rows neither columns, thus, they form a stratum to run in parallel. Once computed, another stratum (2,6,7 or 3,4,8) can run.

3.2. Distribution: paradigm and flexibility.

The main idea behind the parallelization of this algorithm is the division of the matrix into *blocks* and *stratums*. A *block* is a batch of ratings that should be computed. It is computed by only one processor, and hence, parameters updates are available within the same processor and there is no dependency problems inside. However, only independent blocks can be computed in parallel. In fact, parallelizable blocks do not share rows neither columns, as a consequence, ratings of different blocks are independent. A *stratum* is defined as a set of independent blocks. Then, inside a stratum there are no dependency problems. Thus, this creates a stratum schedule to follow, where stratums can be computed in sequence or arbitrary sequence [3]. Then, SGD is stratified and safely parallelizable. Figure 2 represents this paradigm.

This method allows to distribute the process of blocks into machines in a cluster. In order to distribute the algorithm, the number of nodes w and thread processors per node t are taken into account. Thus, the number of single process units is $b = w \cdot t$ (supposing similar node machines). Looking for b independent blocks in the matrix maximizes the efficiency of the distribution in the cluster. However, this may cause a problem of flexibility to the cluster. Imagine a matrix of dimension n_u x n_i, the key point in the block decomposition is the divisibility of the dimensions in integer parts. That is, dimensions might be divisible by b in order to properly create blocks ($n_u \% b = 0$ and $n_i \% b = 0$). This integer divisibility constraint creates a conflict while decomposing matrices in blocks, since dimensions may not verify this "condition".

4. FLEXIBLE DISTRIBUTED STOCHASTIC GRADIENT DESCENT

Our work focuses on coping this divisibility condition. In [3, 16], the deletion of rows/column (i.e. users/items) is suggested to achieve it. However, this may yield in a loss of accuracy: if a significant number of data is deleted the system may loose important information. In fact, this shows an accuracy/cluster flexibility trade-off: more clusters might lead in more deletions, hence an accuracy loss.

Our contribution is flexible in the condition accomplishment. On the one hand, deleting dimensions loses data. On the other hand, adding dimensions may increase the time-consumption of the algorithm. However, this does not yields in a loss of data, since inserting empty dimension does not add any noisy information.

Based on these ideas, we have developed a Flexible Distributed Stochastic Gradient Descent. It uses three modes of usage regarding the available processing nodes and the matrix dimensions: (1) under dimensionality, (2) upper dimensionality, and (3) flexible dimensionality. The under dimensionality is the proposition given by [3]. This mode resize the matrix by deleting dimensions, and thus resized dimension are equal or minor to the real one to accomplish the decomposing condition. The upper dimensionality resizes the matrix by inserting empty dimensions until accomplishing the decomposing condition. Finally, the flexible dimensionality technique is a hybrid of both techniques. It looks for the closest dimension (above or below) to the real one. This technique balances the trade-off. Any of these techniques return new dimensions, n'_u and n'_i, that fits properly with a matrix block decomposition.

4.1. DSGD Hadoop/MapReduce

In this section, we present the steps of our implementation in MapReduce. Three parameters have to be set: row original matrix dimension n_r, column original matrix dimension n_c, and the number of units b. There are three main points in our implementation:

(1) Matrix block decomposition: it adapts the dataset to the number of units of the cluster. To accomplish the decomposition condition and make the rating matrix decomposable in blocks, this method will delete entries and/or add empty entries. Once the decomposing condition is solved, block sizes can be defined. The row step size of a block is given by $rowStep = n'_r/b$ and the column step size is given by $columnStep = n'_c/b$, where n'_r and n'_c are the new dimensions. Hence, the blocks can be formed by moving through the matrix.

(2) Stratum assignment: it groups independent blocks in the same stratum. Note that, the number of the stratums is actually the number of nodes b, and thus, up to b blocks are in the same stratum and can be run in parallel.

(3) Stratum execution: This step computes SGD in stratums blocks and then updates parameters at the end of their execution. This guarantees the integrity and independence of parameters for next stratums. The stratum are sequentially executed, yet this sequence may be randomly created. Finally, the stratums sequence is run a defined number of iterations or until convergence/achieved accuracy.

Our approach uses a $SGDJob$ in MapReduce that analyses multiple entries. These are the blocks of a stratum, which contain the ratings to execute Gradient Descent optimization. The output of the job consist of vectors containing the last updated parameters for users and items. This process is done with a single $Map-function$. The input key is a pair of val-

ues which represent the entries of the matrix, the input value is the rating itself. After computation and a clean-up map process, the map function can return the results. The output key are the row/column of the matrix and a vector representing the updated parameters.

5. EXPERIMENTATION

We assess the performance of our implementation by using the publicly available MovieLens 10M dataset [1], which has 10 million ratings applied by 69878 users to 10667 movies. We create a matrix of rows (users) and columns (items) that contains available ratings. The evaluation and comparisons are carried in terms of accuracy and scalability. On the one hand, the dataset is split up into a training set and a test set, containing respectively the 90% and the 10% of the ratings. These tests allow to evaluate the accuracy by using Root Mean Square Error (RMSE). This metric computes the square error in the difference between predicted values and real-observed values, the lower the better. On the other hand, the time-consuming and scalability parameters depend on machines and clusters. We vary the number of nodes in the cluster. That is, we use $b = 1, 2, 5, 7, 15$. In this paper we run the experimentation in single node cluster: a MacOS 4Go RAM with 2 cores (2.53GHz). This makes mappers in mapreduce to be computed sequentially, thus time-consumption is higher than in multi-cluster. Amazon EMR[2] instances are used for real cluster experimentation, yet results are still on work and are not shown in this paper.

Three methods are compared: under dimensionality (underDSGD), upper dimensionality (upperDSGD), and flexible dimensionality (flexDSGD). These techniques have been equally trained: the overfitting parameter and learning rate were obtained by cross-validation: $\lambda = 0.025$ and $\gamma = 0.0075$. In addition, the algorithms run for 30 iterations and use 30 latent features. Henceforth, it is expected that upperDSGD obtains slightly better results in terms of RMSE, since it does not delete data. Yet, underDSGD might get slightly better time performance. FlexDSGD will show intermediary results in both comparisons. Moreover, we offer a comparison to ALS that shows the superiority of DSGD. The technique uses $\lambda = 0.025$, value obtained by cross-validation. 30 iterations over 30 latent features were performed as well.

Table 1 shows the comparisons and results. The performance in RMSE and the time achieved in computing one iteration in a *single node cluster* are presented. Besides, it shows the number of rows/columns (users/items) that has been added or removed (denoted by "+" or "-", respectively) in order to achieve the divisibility "condition": $n_u \% b = 0$ and $n_i \% b = 0$.

As it was expected, DSGD techniques are more accurate (with a lower RMSE) and faster than ALS. In addition, upperDSGD and flexDSGD slightly overcome to underDSGD in accuracy. Indeed, these two modes deal with more ratings, what allows achieving (tiny 1%) better results. This fact explains as well the little extra time taken in computation. Nevertheless, less or none information is deleted. For instance, focus in the case of 15 number of units, which has 15 independent blocks per stratum. In order to achieve this decomposition, underDSGD has deleted 8 rows (users) and 2 columns (items), yielding in a loss of data. In addition, the deletion is not controlled, and thus one may delete very informativeness rows and columns.

Table 1. Experimentation results.

Technique	RMSE	Time (min)	N. Units (b)	Rows	Columns
ALS [7, 8]	0.79603	3.019	1	=	=
Under DSGD [3]	0.77571	1.303	1	=	=
Flex DSGD	0.77571	1.303	1	=	=
Upper DSGD	0.77571	1.303	1	=	=
Under DSGD [3]	0.77611	1.850	2	=	-1
Flex DSGD	**0.77555**	1.852	2	=	+1
Upper DSGD	0.77559	1.885	2	=	+1
Under DSGD [3]	0.77626	5.518	5	-3	-2
Flex DSGD	0.77617	5.524	5	+2	-2
Upper DSGD	**0.77597**	5.525	5	+2	+3
Under DSGD [3]	0.77586	8.978	7	-4	-6
Flex DSGD	**0.77548**	8.985	7	+3	+1
Upper DSGD	0.77565	9.024	7	+3	+1
Under DSGD [3]	0.77593	28.506	15	-8	-2
Flex DSGD	0.77596	28.526	15	+7	-2
Upper DSGD	**0.77555**	28.536	15	+7	+13

6. CONCLUSION

Recommender Systems aim at selecting and presenting first the information in which users may be more interested. Matrix Factorization recommendation technique achieves high accuracy and scalability in very large datasets. In this paper we focused on a Distributed Stochastic Gradient Descent (DSGD) algorithm based on this technique. It has higher performance compared to other algorithms; yet its distribution paradigm does not fit properly the cluster, what may cause light loss in accuracy.

This paper details an implementation of Distributed Stochastic Gradient Descent in a real Hadoop environment. In addition, we have proposed a slight modification to (1) adapt the distribution to the cluster set up, and (2) avoid the minor loss of accuracy of the current distribution paradigm. The experimentation phase uses the public MovieLens dataset. The evaluations show the good performance of the approach in terms of accuracy and scalability. Future work focuses on enhancing and improving this implementation by incorporating more heterogeneous data to the process.

Acknowledgments

This work has been supported by FIORA project, and funded by "DGCIS" and "Conseil Regional de l'Île de France".

[1] http://grouplens.org/datasets/movielens/

[2] http://aws.amazon.com/elasticmapreduce/

7. REFERENCES

[1] Paul B Kantor, Francesco Ricci, Lior Rokach, and Bracha Shapira, "Recommender systems handbook," in *Recommender systems handbook*. Springer, 2011, p. 848, Springer US.

[2] Neil Hunt, "Recsys 2014. the value of better recommendations," RecSys 2014 Keynote. Slideshare.net, October 2014.

[3] Faraz Makari, Christina Teflioudi, Rainer Gemulla, Peter Haas, and Yannis Sismanis, "Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion," *Knowledge and Information Systems*, pp. 1–31, 2014.

[4] Yehuda Koren, "The bellkor solution to the netflix grand prize," *Netflix prize documentation*, 2009.

[5] Yehuda Koren, Robert Bell, and Chris Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[6] Yehuda Koren and Robert Bell, "Advances in collaborative filtering," in *Recommender Systems Handbook*, pp. 145–186. Springer, 2011.

[7] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen, "Collaborative filtering recommender systems," in *The adaptive web*, pp. 291–324. Springer, 2007.

[8] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Algorithmic Aspects in Information and Management*, pp. 337–348. Springer, 2008.

[9] István Pilászy, Dávid Zibriczky, and Domonkos Tikk, "Fast als-based matrix factorization for explicit and implicit feedback datasets," in *Proceedings of the fourth ACM conference on Recommender systems*. ACM, 2010, pp. 71–78.

[10] Gábor Takács and Domonkos Tikk, "Alternating least squares for personalized ranking," in *Proceedings of the sixth ACM conference on Recommender systems*. ACM, 2012, pp. 83–90.

[11] Prateek Jain, Praneeth Netrapalli, and Sujay Sanghavi, "Low-rank matrix completion using alternating minimization," in *Proceedings of the 45th annual ACM symposium on Symposium on theory of computing*. ACM, 2013, pp. 665–674.

[12] Simon Funk, "Netflix update: Try this at home (3rd place)," 2006.

[13] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk, "Major components of the gravity recommendation system," *ACM SIGKDD Explorations Newsletter*, vol. 9, no. 2, pp. 80–83, 2007.

[14] Yehuda Koren, "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 426–434.

[15] Yehuda Koren, "Factor in the neighbors: Scalable and accurate collaborative filtering," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 4, no. 1, pp. 1, 2010.

[16] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 69–77.