Low-rank matrix factorization using distributed SGD in Spark

Nikhil Parthasarathy and Pin Pin Tea-mangkornpan

Abstract

Obtaining low-rank factorizations of large matrices is an important problem in many fields and is extremely important for problems in machine learning such as collaborative filtering. In this project, we implement a version of an existing distributed stochastic gradient descent (DSGD) algorithm for matrix factorization in the Apache Spark framework. While the basic algorithmic form is not novel, we explore implementation challenges and issues in the context of Spark specific data structures and communication patterns. Two practical comparisons of runtimes for factorizations with various latent factor ranks are made between: 1) our implementation and a sequential SGD method and 2) our implementation and the existing Spark MLlib implementation of alternating least squares (ALS) for matrix factorization. We find that there is significant improvement over the sequential method. However, it is the case that unless both the rank of the factors is extremely large and the data is dense, the ALS method in general converges with a much faster runtime.

1 Introduction

1.1 Problem Statement

The problem of finding low-rank matrix factorizations is one that is extremely important and widely used in many fields. The abstract idea can be described as follows. Consider any matrix $\mathbf{V} \in \mathbb{R}^{n \times m}$. If both n and m are very large, often times, finding a low-rank approximation can be useful for reducing dimensionality, extracting signal from noise, etc. More formally, in finding a low-rank approximation, we seek matrices (or "factor") $\mathbf{W} \in \mathbb{R}^{n \times k}$ and $\mathbf{H} \in \mathbb{R}^{m \times k}$ such that we minimize a loss between the reconstructed matrix and the original $\mathbf{V}$. A common method is to try and find $\mathbf{W}$ and $\mathbf{H}$ such that we minimize the Frobenius norm of the difference:

$$\min \frac{1}{2} \left\| \mathbf{V} - \mathbf{W}\mathbf{H}^{\intercal} \right\|_F^2 \tag{1}$$

If we find a solution to this minimization problem then we have found an approximation for $\mathbf{V}$ that is of rank k matrix (and we hope that we can achieve a reasonable approximation where $k \ll rank(\mathbf{V})$). This problem formulation appears in many places, one of the most common areas being collaborative filtering for recommender systems.

Because the experiments in this work were run on movie recommendation data, for the rest of this paper, we constrain our interpretation of the data to be

as follows. $\mathbf{V}$ is the ratings matrix with rows corresponding to users, columns corresponding to movies and each entry (i, j) is the rating of movie j by user i. $\mathbf{W}$ is the user factor matrix with rows corresponding to each user. $\mathbf{H}$ is the movie factor matrix with rows corresponding to each movie.

1.2 Objective

Many algorithms for solving this problem have been developed both for use on single machines and in a distributed setting. Two of the most prominent distributed algorithms for matrix factorization are the distributed stochastic gradient descent (DSGD) method proposed by Gemulla et al. in [1] and the alternating least squares (ALS) method proposed by Zhou et al. in [5]. There are many frameworks that have been developed for distributed processing but as of now, it seems that the Apache Spark framework is becoming more of a standard in industry. As a result, libraries like the MLlib library are starting to be developed to implement and package useful algorithms for the community to use. Currently, the MLlib library contains only the ALS method for handling low-rank matrix factorization in the collaborative filtering setting. Even though the DSGD algorithm has published implementations on Hadoop based systems, there is no published implementation using Spark. The work of Li et al. in [2] describe an implementation at a high-level in Spark, but they primarily focus on developing a novel data abstraction outside of the current Spark framework and do not provide any published code or benchmarks. Therefore, the goal of this project is as follows:

1. Implement the algorithm of [1] on Spark using only the existing data structures and communication patterns.
2. Benchmark this implementation against the current ALS standard and against a sequential SGD baseline.
3. Analyze the Spark specific challenges and costs in terms of network communication, memory, and computation.

1.3 Outline

The outline of the paper is as follows. Section 2 describes the theory behind the three algorithms we implement. Section 3 discusses the Spark specific implementation details of the DSGD algorithm and provides analysis of the relevant costs. Section 4 provides various benchmark results comparing DSGD to the sequential SGD and to the ALS implementation. Finally, Section 5 provides some final discussion and future work.

2 Algorithms

2.1 Sequential SGD

One of the most basic approaches to solving the optimization problem given in Eq. 1 is to simply iterate sequentially over every non-zero element in the data matrix using SGD. For every training point in the training data, we compute an

L2-loss with respect to the corresponding row and column in the factor matrices. The losses are defined as follows (including a regularization term):

$$\frac{\partial}{\partial \mathbf{W}_{i,:}} L(\mathbf{V}_{ij}, \mathbf{W}_{i,:}, \mathbf{H}_{:,j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i,:}\mathbf{H}_{:,j})\mathbf{H}_{:,j} + 2\lambda \mathbf{W}_{i,:}^{\intercal} \tag{2}$$

$$\frac{\partial}{\partial \mathbf{H}_{:,j}} L(\mathbf{V}_{ij}, \mathbf{W}_{i,:}, \mathbf{H}_{:,j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i,:}\mathbf{H}_{:,j})\mathbf{W}_{i,:}^{\intercal} + 2\lambda \mathbf{H}_{:,j} \tag{3}$$

The algorithm is thus given by the following procedure:

Algorithm 1 Sequential Stochastic Gradient Descent for Matrix Factorization

1: **procedure** SGD
2: $\mathbf{V} \leftarrow$ randomly shuffled training data
3: $W_0 \leftarrow$ randomly initialized W
4: $H_0 \leftarrow$ randomly initialized H
5: **while** not converged **do**
6: Select a training point with indices (i, j) from $\mathbf{V}$
7: $\mathbf{W}^{prev} = \mathbf{W}_0$
8: $\mathbf{H}^{prev} = \mathbf{H}_0$
9: $\mathbf{W}_{i,:} \leftarrow \mathbf{W}_{i,:}^{prev} - \epsilon_n \frac{\partial}{\partial \mathbf{W}_{i,:}^{prev}} L(\mathbf{V}_{ij}, \mathbf{W}_{i,:}^{prev}, \mathbf{H}_{:,j}^{prev})$
10: $\mathbf{H}_{:,j} \leftarrow \mathbf{H}_{j,:}^{prev} - \epsilon_n \frac{\partial}{\partial \mathbf{H}_{:,j}^{prev}} L(\mathbf{V}_{ij}, \mathbf{W}_{i,:}^{prev}, \mathbf{H}_{:,j}^{prev})$

Here ϵ_n is the step size.

2.2 DSGD

Looking at the sequential SGD, we see that the update is dependent on row i of $\mathbf{W}$ and column j of $\mathbf{H}$ and the entry $\mathbf{V}_{ij}$. The key insight into distributing the SGD update centers is based on the concept of "interchangeability". Consider two sets of row indices I and I' of $\mathbf{V}$. Also define sets of column indices J and J'. Now two matrix blocks $I \times J$ and $I' \times J'$ are interchangeable if $I \cap I' = \emptyset$ and $J \cap J' = \emptyset$. If we consider a matrix block $\mathbf{V}_{IJ}$, then the gradient updates determined by $\{\mathbf{V}_{IJ}, \mathbf{W}_{I,:}, \mathbf{H}_{J,:}\}$ and $\{\mathbf{V}_{I'J'}, \mathbf{W}_{I',:}, \mathbf{H}_{J',:}\}$ do not have any overlapping computations, so they can be distributed in parallel. Therefore, in the DSGD method, the main goal is to find sets of interchangeable blocks (known as *strata*) so that each block within a stratum can be updated in parallel using SGD. As long as the parameters have not converged, we keep selecting a new stratum and feed the blocks of the stratum to the separate workers. For illustrative purposes, we show Figure 1 taken from [1] that presents a possible set of strata that covers a data matrix blocked into a three by three grid: We

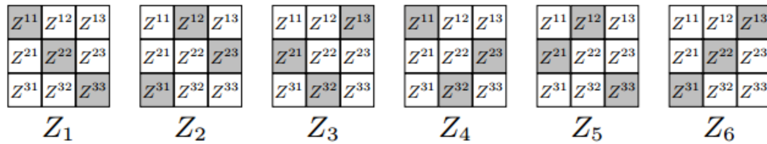

Figure 1: Strata for a 3 × 3 blocking of training matrix $\boldsymbol{Z}$

can thus think of the overall loss function as being decomposed into a linear combination of local losses associated with each stratum:

$$L(\mathbf{W}, \mathbf{H}) = \sum_{s=1}^{q} w_s L_s(\mathbf{W}, \mathbf{H}) \tag{4}$$

w_s is a weight associated with each stratum specific loss. For the purposes of this project, we set $w_s = \frac{N_s}{N}$ where N_s is the number of non-zero in the stratum and N is the total number of non-zero entries. For the specific DSGD algorithm details and structure, see Section 3.

2.3 ALS

Finally, as we are benchmarking our DSGD implementation against the Spark implementation of ALS for matrix factorization, we briefly provide an overview of the ALS method. The ALS procedure can be described by the following steps:

1. Randomly initialize the matrix $\mathbf{H}$ by assigning the average rating for the corresponding movie (in the case of movie recommendation) as the first row, and add small random numbers for the remaining entries
2. Fix $\mathbf{H}$ and solve for $\mathbf{W}$ that minimizes the objective function (the root mean square error (RMSE)).
3. Fix $\mathbf{W}$ and solve for $\mathbf{H}$ that minimizes the objective function similarly.
4. Repeat steps 2 and 3 until convergence.

The details of how to minimize the objectives etc. are given in [5], but we note a few key ideas that are specific to distributing ALS. First, two distributed copies of the ratings matrix $\mathbf{V}$, one distributed by rows and one distributed by columns must be broadcast to each machine. The updating of $\mathbf{W}$ and $\mathbf{H}$ also require replicating and communicating the corresponding factor matrix during each update. Users and movies must be partitioned into groups and the updates corresponding to specific groups are sent to separate machines which are then gathered after the computation is done. While this description provides a high-level view of the ALS method, the Spark specific implementation details (which are necessary to do a full theoretical comparison of DSGD with ALS) are not analyzed in this project.

3 Implementation

3.1 Our Design

We implemented DSGD using Spark (in Pyspark). All code for this project can be found at `https://github.com/pinnareet/CME323DSGD`. Our pseudocode is depicted in Algorithm 2. First, the data is parsed and stored as a coordinate matrix $\mathbf{V}$ (Line 2), which we use its dimension to initialize the factor matrices $\mathbf{W}$ and $\mathbf{H}$ (Line 4-5). We initialize $\mathbf{W}$ and $\mathbf{H}$ as RDDs with each entry of $\mathbf{W}$ corresponding to each row, and each entry of $\mathbf{H}$ corresponding to each column (Line 6-7). Every each entry is keyed with its original row and column index. Note that with our algorithm, we can fix $\mathbf{W}$'s row index as key, and permute

H's column index to obtain a stratum. Therefore, we assign each entry of **W** with its row block index before performing the DSGD iterations (Line 8). Each stratum is processed sequentially until the factors converge. In the loop, we pick a stratum, preassigned by fixing the row index and permuting the column index. Note that this guarantees non-overlapping blocks of matrices. Then we filter **V**, as well as change it into an RDD so that the RDD contains only entries of **V** that correspond to the selected statum (Line 11). Each entry of RDD is keyed with its row block index, which is sufficient given that we have already filtered the columns of **V** (Line 12). Next, we key **H** with its corresponding permuted column index (Line 13). Designed to optimize distributed computing, Spark abstracts users from specifying the partitions in its basic transformation interface. We need to override Spark's automatic partition to ensure that corresponding blocks of **V**, **W**, and **H** will be computed on the same processor. We combine the **V**, **W**, and **H** RDDs using the Spark transformation *cogroup*() (a.k.a. *groupWith*()), which returns an RDD of a tuple (k, combined RDDs of **V**, **W**, and **H** that has key k) (Line 14). Then we use *partitionBy*() to ensure that each partition of *cojoinedRDD* will be run on the same processor (Line 15). Finally, we map each partition of *cojoinedRDD* using *mapPartitions*(), which runs SGD on each block in the stratum in parallel (Line 16).

Algorithm 2 Distributed Stochastic Gradient Descent for Matrix Factorization

1: **procedure** DSGD
2:     $\mathbf{V} \leftarrow$ training data
3:     $rank \leftarrow$ number of latent factors
4:     $numRows \leftarrow$ number of rows of **V**
5:     $numCols \leftarrow$ number of columns of **V**
6:     $\mathbf{W} \leftarrow$ randomly initialized matrix with dimension $numRows \times rank$
7:     $\mathbf{H} \leftarrow$ randomly initialized matrix with dimension $rank \times numCols$
8:     $keyedW \leftarrow$ a tuple of each entry of **W** and its corresponding block number
9:     **while** not converged **do**
10:         Select a stratum
11:         $VF \leftarrow$ filtered matrix **V** to the corresponding stratum
12:         $keyedVF \leftarrow$ a tuple of each entry of VF with each block keyed with its corresponding block number
13:         $keyedH \leftarrow$ a tuple of each entry of **H** and its corresponding permuted block number
14:         $cojoinedRDD \leftarrow VF$, $keyedH$, and $keyedW$ grouped by key
15:         Assign each partition of *cojoinedRDD* with the same key to be distributed to the same processor
16:         Run SGD on each partition of *cojoinedRDD* in parallel

3.2 Design Choices

Instead of shipping **W**, **H**, and a stratum of **V** around when providing each block data to each processor, we can alternatively broadcast **V**, **W**, and **H** to every processor and perform DSGD with less communication cost. For **V**, broadcasting the matrix may be more efficient if the total number of entries is

small compared to available memory space, since we do not have to communicate the stratum of $\mathbf{V}$ at every iteration of stratum selection. For $\mathbf{W}$ and $\mathbf{H}$, however, one needs to broadcast the entire matrices and still update the matrices at every SGD iteration. We analyze the communication cost between these design choices in the following subsection.

3.3 Analysis

Let $\mathbf{V} \in \mathbb{R}^{m\times n}$, $\mathbf{W} \in \mathbb{R}^{m\times r}$, $\mathbf{H} \in \mathbb{R}^{r\times n}$, $\mathbf{V}$ has q entries, the number of workers/processors be k, and the number of iterations be i. On the one hand, if we broadcast all matrices, we need to broadcast the entire $\mathbf{V}$, $\mathbf{W}$, and $\mathbf{H}$ only once at the beginning. The communication cost is $O(k(q + mr + nr))$, because we all entries of $\mathbf{V}$, and the entire matrices $\mathbf{W}$ and $\mathbf{H}$ to k workers. Moreover, we need to update $\mathbf{W}$ and $\mathbf{H}$ at every iteration. Note that we need only update the stratified blocks of $\mathbf{W}$ and $\mathbf{H}$, so the all-to-one communication takes $O(k\frac{mr+nr}{k}) = O(mr + nr)$ for each iteration. Therefore, the total communication cost is $O(kq + k(mr + nr) + i(mr + nr)) = O(kq + (k + i)(mr + nr))$, which would be $O(k(q + mr + nr))$ if i scales linearly (or less) with k, and $O(kq + i(mr + nr))$ otherwise.
On the other hand, our implementation sends the filtered blocks of $\mathbf{V}$, $\mathbf{W}$, and $\mathbf{H}$ to each worker at every iteration. For a single iteration, the expected communication cost is $O(k\frac{q}{k^2}) = O(\frac{q}{k})$ for shipping filtered entries of $\mathbf{V}$ to k workers, and $O(k\frac{(mr+nr)}{k}) = O(mr + nr)$ for shipping a block of $\mathbf{W}$ and $\mathbf{H}$ to k workers. Note that the expected value of entries of V in each block of a stratum is $\frac{q}{k^2}$ because we perform *data-independent blocking*, which ensures that the expected number of training points in each block of $\mathbf{V}$ is $\frac{q}{k^2}$ [1]. Hence, the total communication cost is $O(i(\frac{q}{k} + mr + nr))$.
Comparing the communication costs of the two approaches, $O(kq+(k+i)(mr+nr))$ and $O(i(\frac{q}{k} + mr + nr))$, there are several scenarios we must consider. If i scales linearly with k, our implementation definitely fares better. If i scales at least quadratically with k, then we need to consider the number of latent factors r, and the sparsity of $\mathbf{V}$. If $\mathbf{V}$ is so dense that the q term dominates, then broadcasting would be a better choice. If the $mr+nr$ term dominates, and given that i at least quadratically larger than k, then our implementation is slightly better (having $k(mr + nr)$ less elements that need to be communicated). Both designs should yield similar performance in big-O time. In our experiment, we do not see poor performance as r gets larger as mentioned in [2] because the dataset is dense, which makes the q term dominates. However, since we do not know how i scales with k, broadcasting does not necessarily yield better performance. In general, i should be independent of k, so we may need to experiment and choose the appropriate design depending on the sparsity structure of the data.

4 Results

The DSGD algorithm was implemented in PySpark and run using a Spark 1.6.1 installation on a DataBricks cluster. The cluster had 13 worker nodes each with 30 GB of memory. The main dataset that was used for the experiments was the MovieLens 1M dataset which consists of 1 million ratings with 6040 users and 3883 movies [3]. In addition, for limited experiments we also utilized

the MovieLens 10M dataset which consists of 10 million ratings with 71567 users and 10681 movies (because of time constraints we were not able to run all experiments with this much larger dataset).

4.1 DSGD Scaling

We first tested the scaling of the DSGD implementation runtime against the number of workers and the rank of the factors **W** and **H**. Figure 2 shows these results.

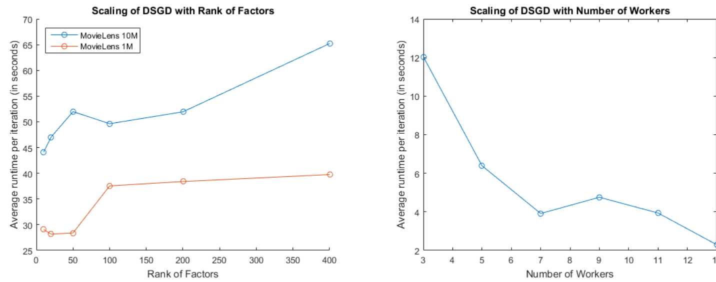

Figure 2

We can see that as expected the runtime per iteration does go up as the rank of the factors gets larger, but even with the large 10 million rating dataset, the increase is not prohibitively significant for rank less than 200. We expect that the effect of the rank of the factors will most likely only become a serious problem when the number of users/movies is much larger. In addition, we see a great speedup per iteration (super-linear) with the increase in number of workers. It is important to note again, however, that these effects are most likely related to the fact that the MovieLens dataset is pretty dense. Therefore, the costs of communicating the factors is dominated by the cost of communicating the actual data blocks.

4.2 SGD vs. DSGD

The first comparison made was between DSGD and sequential SGD. We simulate sequential SGD by running our implementation on a single machine with one worker. Figure 3 shows the runtime of DSGD and sequential SGD with factor rank 50. Notice that DSGD converges approximately 4 times faster than sequential SGD implementation in runtime. DSGD's MSE drops significantly within 30 seconds, whereas that of SGD reaches the same MSE at more than 150 seconds. The result confirms that DSGD has significant performance advantage over sequential SGD.

4.3 ALS vs. DSGD

The main goal of this project was to see if the DSGD algorithm could implemented in Spark to be competitive with the current ALS implementation. Because we could not get into the code of the ALS algorithm to print values at every iteration, we report comparisons of the final runtimes required by each

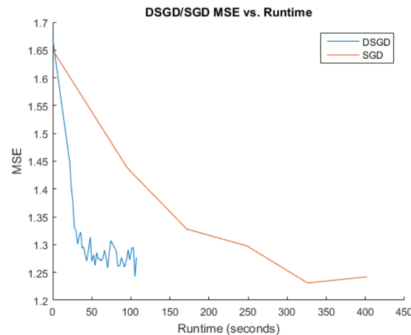

Figure 3

method to reach convergence in the factors. Since we were using the out-of-the-box ALS implementation, we could not retrieve convergence results, however, the documentation suggests that running the algorithm for 20 iterations will produce reasonable results. Therefore, we run the ALS for 20 iterations and compare to the DSGD convergence time [1]. Both implementations were run on the MovieLens 1M dataset on the same cluster.

Factor Rank	ALS	DSGD
10	20.1 s	51.0 s
20	20.0 s	45.6 s
50	15.7 s	53.4 s
100	23.5 s	55.5 s
200	38.7 s	58.7 s
400	130.1 s	63.6 s

Table 1: Comparison of ALS and DSGD runtimes to convergence for different ranks of factors

Interestingly, even though the ALS out-performs the DSGD implementation for all ranks below 200, for rank of 400, the DSGD actually converges faster than ALS. This indicates that for dense matrices, DSGD might scale better with the rank of factors than the ALS algorithm. In order to see how these two methods compared with scaling of the data size, we ran ALS and DSGD on the MovieLens 10M dataset, setting the rank of the factors to a reasonable size of $k = 50$. In this case, the ALS ran in 108 seconds whereas the DSGD ran in 642 seconds, indicating that the current DSGD implementation scales far worse than the ALS implementation with large scaling of the number of data entries. We expect this difference to be mitigated, however, if we had many more workers.

5 Discussion

We have implemented DSGD in Apache Spark, and compared runtimes of our implementation with sequential SGD and the Spark MLlib implementation of

[1]DSGD convergence achieved when the MSE stops changing within a fixed tolerance.

ALS for matrix factorization. For DSGD implementation, we choose to communicate the stratified data matrix and the factor matrices as opposed to broadcasting the variables. We have shown that the communication cost of both approaches are $O(i(\frac{q}{k} + mr + nr))$ and $O(kq + (k+i)(mr + nr))$ (the variables are defined in Section 3.3), respectively, which makes our implementation a better choice if the data is not too sparse and the number of iterations does not scale badly with the number of rank factors. Our experiment with the MovieLens dataset shows that for DSGD, the runtime per iteration increases with rank, but the magnitude is not significant for ranks less than 200. We also see significant increase in per iteration speed with the increase of number of workers. Comparing DSGD with sequential SGD and ALS, we found that DSGD outperforms SGD in by 4-fold in runtime for rank 50, but outperforms ALS only when the rank is 400 or greater. However, DSGD scales worse than ALS with data size, which we expect to mitigate by increasing the number of workers.
In future work, we plan to run experiments with our implementation on larger datasets, such as the Netflix Prize dataset, and implement the algorithm in Scala. As discussed in 3.3, we may need to change the implementation depending on the sparsity of the matrix. We also plan to investigate the SparkMLlib ALS implementation, as well as perform theoretical analysis comparing it to DSGD. Finally, we plan to explore Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion (NOMAD), an asynchronous parallel distributed algorithm for matrix completion . Yun et al. claims that NOMAD outperforms DSGD on all configurations of their experiment [4]. Our current work could not be compared to NOMAD because we did not obtain the same datasets.

References

[1] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. *Conference on Knowledge Discovery and Data Mining*, pages 69–77, 2011.

[2] B. Li, S. Tata, and Y. Sismanis. Sparkler: Supporting large-scale matrix factorization. *EDBT*, page 625–636, 2013.

[3] GroupLens Research. MovieLens. http://grouplens.org/datasets/movielens/. Accessed: 2016-06-01.

[4] H. Yun, H. Yu, C. Hsieh, S. Vishwanathan, and I. Dhillon. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. *arXiv preprint arXiv:1312.0193*, 2013.

[5] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. *Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, page 337–348, 2008.