Parallel Online Collaborative Filtering for Streaming Data

Muqeet Ali Christopher C. Johnson Alex K. Tang

December 7, 2011

Abstract

We present a distributed stochastic gradient descent algorithm for performing low-rank matrix factorization on streaming data. Low-rank matrix factorization is often used as a technique for collaborative filtering. As opposed to recent algorithms that perform matrix factorization in parallel on a batch of training examples [4], our algorithm operates on a stream of incoming examples. Assuming sufficient conditions are met, we show that our algorithm finds a locally optimal factorization in significantly less time than standard non-distributed approaches. We further corroborate these results with an experimental comparison to a state-of-art-method in batch low-rank matrix factorization.

1 Introduction

The problem of collaborative filtering, or predicting the interests of a user by collectively filtering the interests of a large group of users, has garnered a considerable amount of attention in recent years as the size and ubiquity of Internet-sized data has grown at an exponential rate. Though there are various formulations to the problem of collaborative filtering, one of the most general is to fill in the unknown entries in an incomplete matrix $R \in \mathbb{R}^{n \times m}$ of user ratings. (Here n is the number of users, m is the number of items, and R_{ij} is the rating that user i assigns to item j.) One recognizable practical domain for this problem comes in the form of the Netflix Prize,[1] in which Internet movie rental company Netflix [2] offered a prize of $1 million to any group that could improve the accuracy of their movie recommendation engine by 10%.

One of the most successful approaches in solving the collaborative filtering problem aims at finding a low-rank matrix factorization of the ratings matrix $R \approx U^T V$, where $U \in \mathbb{R}^{k \times n}$, $V \in \mathbb{R}^{k \times m}$, $k \ll m$, $k \ll n$. Standard SVD factorization is undefined for a matrix containing missing data, and methods to bypass this using SVD factorization have generally been unsuccessful. Additionally, finding an optimal matrix factorization in general involves a non-convex optimization. One method of circumventing this drawback is that of alternating least squares (ALS), in which each low-rank matrix is alternatingly held fixed while the other is optimized using least-squares regression or ridge regression (where a regularization term is added to the square loss to avoid overfitting). Another group of methods for low-rank matrix factorization use stochastic gradient descent (SGD) [3]. These methods choose ratings at random and for each rating update a column of U and a column of V by taking a small step in the direction opposite the gradient of the loss function at that specific point.

The size of data being used in collaborative filtering is often very large due to the ease of recording Web-scale user interests. Furthermore, it has been shown in many learning domains that using larger data sets will increase performance far beyond that of constructing better learning algorithms [5]. For these reasons, methods of computing low-rank matrix factorizations using large scale data sets, the size of which can often not fit on a single machine, are becoming increasingly important in practical domains. With the recent advent of parallel processing frameworks such as MapReduce [2] (for batch data) and Storm [8] (for streaming data), dealing with large data sets has become more feasible. Zhou et al. [10] describe a parallelized implementation of ALS that they wrote using parallelized MATLAB, while Sebastian Schelter has written a parallelized implementation of ALS for the Apache Mahout project [9] using the Hadoop MapReduce framework. Like ALS, the SGD algorithm for matrix factorization has also been parallelized: Gemulla et al. [4] describe a parallelized implementation of SGD written using MapReduce. The algorithm of Gemulla

[1] `http://www.netflixprize.com`
[2] `http://www.netflix.com`

et al. operates on batch data. We describe an approach for performing low-rank matrix factorization on streaming data. In this setting, we begin with an empty set of data points. Data points then stream in in an online manner and can be stored if needed. The goal is to then begin processing data points as they stream in, before receiving the full set of data points.

2 Preliminaries

Suppose we have n users who assign ratings to m items. We shall assume that ratings are positive real numbers. We can represent the ratings as a matrix $R \in \mathbb{R}^{n\times m}$, where R_{ij} is the rating that user i assigns to item j. Not every user will rate every item; if user i does not rate item j, then we set R_{ij} to zero. The number of items m will typically be very large, so each user will rate only a small fraction of the m items. Hence, we expect the matrix R to be sparse, that is, have few non-zero entries. For each entry (i, j) with $R_{ij} = 0$, we assume that there is some unknown rating that user i assigns to item j. Our goal is to predict these unknown ratings.

We assume that the ratings users assign to items are determined by k latent attributes. Each item j has k weights $v_{j1}, \ldots, v_{jk} \in \mathbb{R}$, one for each latent attribute; we can write these weights succinctly as a vector $v_j = [v_{j1} \cdots v_{jk}]^T$. Similarly, each user i has vector of k weights $u_i = [u_{i1} \cdots u_{ik}]^T$, perhaps representing the importance that user i assigns to each of the latent variables. We assume that the rating user i assigns to item j is given by the inner product

$$\langle u_i, v_j \rangle = u_i^T v_j = u_{i1}v_{j1} + \cdots + u_{ik}v_{jk}.$$

Let the matrices $U \in \mathbb{R}^{k\times n}$ and $V \in \mathbb{R}^{k\times m}$ be defined as $U := [u_1 \cdots u_n]$ and $V := [v_1 \cdots v_m]$. Then, the matrix $U^T V$ consists of all the ratings that the users assigns to items. The number of latent attributes k is typically chosen to be much smaller than both n and m. Thus, storing matrices U and V requires much less space than storing $U^T V$: it takes $O(nm)$ space to store $U^T V$, but only $O(k(n + km))$ space to store both U and V. Our goal is to find a matrix close to R that can be written as the product of two matrices U^T and V of rank at most k (the meaning of "close" is made precise below). This problem is known as *low-rank matrix factorization*.

Let K denote the set of entries in R for which the rating is known; that is,

$$K := \big\{(i, j) \in [n] \times [m] \;\big|\; R_{ij} \neq 0\big\}.$$

We wish to find matrices $U \in \mathbb{R}^{k\times n}$ and $V \in \mathbb{R}^{k\times m}$ which minimize the sum

$$\sum_{(i,j)\in K} \big(R_{ij} - (U^T V)_{ij}\big)^2 = \sum_{(i,j)\in K} (R_{ij} - u_i^T v_j)^2.$$

In other words, the difference between the known ratings and their corresponding predicted ratings should be small. To prevent overfitting, we introduce a regularization term with a penalty parameter $\lambda > 0$. Thus, our objective becomes

$$\underset{\substack{U\in\mathbb{R}^{k\times n}\\ V\in\mathbb{R}^{k\times m}}}{\text{minimize}} \quad \sum_{(i,j)\in K} \big(R_{ij} - (U^T V)_{ij}\big)^2 + \lambda\|U\|_F^2 + \lambda\|V\|_F^2. \tag{1}$$

Here $\|\cdot\|_F$ denotes the Frobenius norm, which is defined by formula $\|A\|_F := \sqrt{\sum_{i,j} A_{ij}^2}$ for each matrix A.

We can rewrite (1) as

$$\underset{\substack{U\in\mathbb{R}^{k\times n}\\ V\in\mathbb{R}^{k\times m}}}{\text{minimize}} \quad \sum_{i=1}^{n}\sum_{j=1}^{m} \left(\mathbb{1}[R_{ij} \neq 0](R_{ij} - u_i^T v_j)^2 + \frac{\lambda}{m}\|u_i\|^2 + \frac{\lambda}{n}\|v_j\|^2\right). \tag{2}$$

In the above formula, we have used the indicator notation $\mathbb{1}[\,\cdot\,]$, which is defined as follows:

$$\mathbb{1}[\mathcal{P}] := \begin{cases} 1 & \text{if } \mathcal{P} \text{ is true,} \\ 0 & \text{if } \mathcal{P} \text{ is false.} \end{cases}$$

Note that (2) is a special case of the more general optimization problem

$$\underset{\substack{U \in \mathbb{R}^{k \times n} \\ V \in \mathbb{R}^{k \times m}}}{\text{minimize}} \quad \sum_{i=1}^{n} \sum_{j=1}^{m} L(R_{ij}, u_i, v_j),$$

where $L : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is a given loss function. For the optimization problem in (2), we use the regularized square loss

$$L(r, u, v) := \mathbb{1}[r \neq 0](r - u^T v)^2 + \frac{\lambda}{m}\|u\|^2 + \frac{\lambda}{n}\|v\|^2. \tag{3}$$

The optimization problem in (1) is not convex, so we cannot directly apply convex optimization methods. Different heuristics have been suggested for solving this optimization problem. One approach is alternating least squares (ALS). If we fix one of the matrices, say U, then the optimization problem simplifies to a ridge regression problem in the other matrix V. The ALS algorithm performs several rounds of ridge regression. During each round, ALS fixes either U or V and optimizes the other matrix—alternating which matrix is fixed every round.

Another approach is stochastic gradient descent (SGD). Each training example is an ordered triple (i, j, R_{ij}) with $i \in [n]$ and $j \in [m]$. The SGD algorithm iterates through all the training examples in random order; for each example (i, j, R_{ij}), it fixes all the columns of U except for u_i and all the columns of V except for v_j, and updates u_i and v_j by taking a small step in the direction opposite the gradient; more precisely,

$$u_i \leftarrow u_i - \eta \nabla_{u_i} L(R_{ij}, u_i, v_j) \qquad \text{and} \qquad v_j \leftarrow v_j - \eta \nabla_{v_j} L(R_{ij}, u_i, v_j), \tag{4}$$

where $\eta > 0$ is the step size. Gemulla et al. [4] and Abernethy et al. [1] show how the SGD algorithm for matrix factorization can be parallelized to distribute the workload among multiple processors. We adapt their approach to an online setting in which the training examples arrive as a stream. Futher details are given in the next section.

3 Distributed Stochastic Gradient Descent

In Section 3.1, we review the distributed stochastic gradient descent (DSGD) algorithm of Gemulla et al. [4]. Essentially the same algorithm appears in an unpublished manuscript by Abernethy et al. [1]. In Section 3.2, we adapt the batch DSGD algorithm to an online setting in which the training examples arrive as a continuous stream of data.

3.1 Batch Algorithm

Here we review the batch algorithm for distributed stochastic gradient descent [1,4]. Consider the regularized square loss function $L : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ defined by (3). The gradient of L with respect to u is given by

$$\nabla_u L(r, u, v) = \mathbb{1}[r \neq 0] \cdot 2(u^T v - r)v + \frac{2\lambda}{m}u, \tag{5}$$

and the gradient of L with respect to v is given by

$$\nabla_v L(r, u, v) = \mathbb{1}[r \neq 0] \cdot 2(u^T v - r)u + \frac{2\lambda}{n}v. \tag{6}$$

Suppose (i, j, R_{ij}) is a training example, or in other words, user i assigns rating $R_{ij} \neq 0$ to item j. By substituting (5) and (6) into (4), we obtain the stochastic gradient descent update rule for the regularized square loss:

$$u_i \leftarrow u_i - 2\eta \left((u_i^T v_j - R_{ij})v_j + \frac{\lambda}{m}u_i \right) \qquad \text{and} \qquad v_j \leftarrow v_j - 2\eta \left((u_i^T v_j - R_{ij})u_i + \frac{\lambda}{n}v_j \right). \tag{$*$}$$

The stochastic gradient descent (SGD) algorithm iterates through all the training examples in random order and applies $(*)$ to each example; the pseudocode is given in Algorithm 1.

Algorithm 1: Stochastic gradient descent for matrix factorization

Initialize U and V randomly.
repeat
  foreach example (i, j, R_{ij}) **do** *// Iterate through the examples in random order.*
    $u_i \leftarrow u_i - 2\eta\big((u_i^T v_j - R_{ij})v_j + (\lambda/m)u_i\big)$
    $v_j \leftarrow v_j - 2\eta\big((u_i^T v_j - R_{ij})u_i + (\lambda/n)v_j\big)$
until convergence

Now, suppose we want to parallelize SGD to distribute the workload among d processors. This can be accomplished by dividing the ratings matrix R into blocks and distributing the blocks to different processors. The problem is that two processors working on different blocks may need to update the same column of U or the same column of V. We must distribute the blocks in a way that avoids conflicting updates. Notice that when processing example (i, j, R_{ij}), Algorithm 1 updates only column u_i and column v_j. Suppose that we partition matrix R into d^2 blocks of size $(n/d) \times (m/d)$, matrix U into d blocks of size $k \times (n/d)$, and matrix V into d blocks of size $k \times (m/d)$ as shown in Figure 1. (For simplicity, we assume that d divides both n and m.) For all $p, q \in [d]$, the examples in block B_{pq} affect only the columns in U_p and the columns in V_q. Thus, if the rows and columns of two blocks B_{pq} and $B_{p'q'}$ do not overlap (i.e., $p \neq p'$ and $q \neq q'$), then the updates made by a processor working on block B_{pq} will not conflict with those made by a processor working on block $B_{p'q'}$. Gemulla et al. [4] call a set of d blocks whose rows and columns don't overlap a *stratum*. The processors can each work on a different block from the same stratum without making conflicting updates. To iterate through all the training examples, we choose a collection of strata which covers the entire ratings matrix R and iterate across the strata, distributing each block in a single stratum to a different processor.

$$R = \begin{array}{|c|c|c|c|} \hline B_{11} & B_{12} & \cdots & B_{1d} \\ \hline B_{21} & B_{22} & \cdots & B_{2d} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline B_{d1} & B_{d2} & \cdots & B_{dd} \\ \hline \end{array} \qquad U = \begin{array}{|c|c|c|} \hline U_1 & \cdots & U_d \\ \hline \end{array} \qquad V = \begin{array}{|c|c|c|} \hline V_1 & \cdots & V_d \\ \hline \end{array}$$

Figure 1: Block partitioning of matrices

Every stratum has the form $\big\{B_{p,\pi(p)} \;\big|\; p \in [d]\big\}$ for some permutation π of the set $[d]$. To cover the entire ratings matrix R, we must choose a set of permutations Π such that

$$\big\{(p, \pi(p)) \;\big|\; p \in [d] \text{ and } \pi \in \Pi\big\} = [d] \times [d]. \qquad (\dagger)$$

For example, one possible choice for Π is the set of permutations generated by the cycle $(2\ 3 \cdots d\ 1)$. But any set of d permutations satisfying (†) can be used.

The preceding discussion is summarized in Algorithm 2. One processor is designated as the master, and d processors are designated as workers. The master distributes the blocks to the workers, and the workers apply the SGD update rule $(*)$ to the examples in the blocks. On each iteration through the training set, the master chooses a set of d permutations satisfying (†) at random.

3.2 Streaming Algorithm

Now, suppose the training examples arrive as a continuous stream. We would like to begin training even if we do not have all the data. We achieve this by modifying the master to assign blocks to workers dynamically rather than assigning the work based on sets of strata. More specifically, the master chooses a free, non-empty block uniformly at random, locks every block in the same row and column, and assigns a worker to process the chosen block. When the worker finishes processing that block, it sends a reply to the master, which then unlocks the blocks in the appropriate row and column.

Storage of the training examples is divided among the workers. Each worker stores only those examples that it needs to process; the master is responsible for routing the examples to the correct worker. Storage of

Algorithm 2: Distributed stochastic gradient descent (batch version)

master:
  Randomly permute the rows and columns of R.
  Initialize U and V randomly.
  repeat
    Randomly choose a set Π of d permutations satisfying (†).
    foreach $\pi \in \Pi$ **do**
      for $p \leftarrow 1$ **to** d **do**
        Give $(B_{p,\pi(p)}, U_p, V_{\pi(p)})$ to worker p.
      repeat
        Wait until $(U_p, V_{\pi(p)})$ is received from a worker p.
        Update the columns of U and V corresponding to U_p and $V_{\pi(p)}$.
      until all workers have responded
  until convergence

worker:
  repeat forever
    Wait until (B_{pq}, U_p, V_q) is received from the master.
    foreach example (i, j, R_{ij}) in block B_{pq} **do** *// Iterate through the examples in random order.*
      $u_i \leftarrow u_i - 2\eta\big((u_i^T v_j - R_{ij})v_j + (\lambda/m)u_i\big)$
      $v_j \leftarrow v_j - 2\eta\big((u_i^T v_j - R_{ij})u_i + (\lambda/n)v_j\big)$
    return (U_p, V_q) to master.

the user matrix U and item matrix V is also divided among several data stores, which we call *matrix stores*. Thus, no process needs to store the entire data set at any point in time.

The streaming algorithm described above is listed as Algorithm 3. Notice that we did not specify a termination condition, because the algorithm is supposed to continuously process an unending stream of data. However, for the purposes of testing, it may be convenient to limit the number of times each training example is processed. In our implementation, we kept a counter for each training example and stopped training when every example was processed a specific number of times.

4 Implementation

4.1 Batch/Hadoop

We implemented the DSGD algorithm using the Hadoop Map-Reduce parallelization framework [3]. A Hadoop program is composed of iterative jobs that consist of 2 phases, a map phase and a reduce phase. The map phase takes as input a set of key-value pairs and outputs a new set of key-value pairs. In the reduce phase, map output key-value pairs with the same keys are merged and set to the same reducer which then outputs a new set of key-value pairs. In this manner, tasks that are parallelizable can be broken down into a set of map and reduce phases and scaled across a cluster of machines.

Our implementation of DSGD consisted of 2 main loops. The outer loop looped over the number of iterations input as a parameter. The inner loop chose an initial starting stratum and then proceeded to perform SGD updates on each subsequent stratum, parallelizing block computations, until the entire ratings matrix was covered. In other words, the inner loop consisted of b tasks, each of which computed b sets of updates in parallel, where b is the number of blocks used. In computing a stratum the map phase took as input the full ratings matrix R and the current factorized matrices U and M. For each individual rating r the mapper first checked if the rating was part of the current stratum. If so, it output a the key value pair $< B_i, r >$ where B_i is the stratum block that r is contained in. For each row vector u and m of U and M the mapper output the key value pairs $< B_i, u >$ and $< B_i, m >$ where B_i is the stratum block that the vector

[3] http://http://hadoop.apache.org/

Algorithm 3: Streaming DSGD

master:

repeat forever

if an example (i, j, R_{ij}) is received **then**

Let B_{pq} be the block to which (i, j, R_{ij}) belongs.

Update bookkeeping information for block B_{pq}.

Forward (i, j, R_{ij}) to worker p.

Let S be the set of unlocked, non-empty blocks.

if $S \neq \varnothing$ **then**

Choose a block B_{pq} from S uniformly at random.

Lock all blocks in row p and all blocks in column q.

Send (p, q) to worker p.

if (p, q) is received from a worker **then**

Unlock all blocks in row p and all blocks in column q.

worker:

repeat forever

if an example (i, j, R_{ij}) is received **then** save (i, j, R_{ij})

if (p, q) is received from the master **then**

Request U_p from user matrix store p.

Request V_q from item matrix store q.

when U_p and V_q have both been received

// Iterate through the examples in random order.

foreach example (i, j, R_{ij}) in block B_{pq} **do**

$u_i \leftarrow u_i - 2\eta\big((u_i^T v_j - R_{ij})v_j + (\lambda/m)u_i\big)$

$v_j \leftarrow v_j - 2\eta\big((u_i^T v_j - R_{ij})u_i + (\lambda/n)v_j\big)$

Send U_p to user matrix store p.

Send V_q to item matrix store q.

Send (p, q) back to the master.

matrix store:

if U_p is requested by a worker **then**

if U_p has not been created **then**

Initialize U_p randomly.

Send U_p to worker.

if U_p is received from a worker **then** save U_p

if V_q is requested by a worker **then**

if V_q has not been created **then**

Initialize V_q randomly.

Send V_q to worker.

if V_q is received from a worker **then** save V_q

belongs to. On the reduce side, all rating items and factor vectors that belong to the same block are sent to the same reducer where SGD updates are iteratively performed. In order to avoid having to buffer all ratings on the reduce side we used the heuristic of secondary sorting, as suggested by Lin/Dyer [6], for which all factor vectors were sorted ahead of all ratings. In this manner, upon receiving a rating we were ensured that we had the necessary factor vectors and could perform the factor updates and discard the rating before looking at the next. Upon completing all SGD updates for its particular block each reducer then output the updated factor vectors. Upon completion of all iterations the final factor matrices U and M were output.

4.2 Streaming/Storm

We implemented our streaming DSGD algorithm using Storm [7, 8], a framework for performing distributed computation on data streams. The primary unit of data in Storm is called a *tuple*, which is just a record containing fields and their values. A *stream* is a (potentially unbounded) sequence of tuples; every tuple in a single stream must have the same fields. A network of processes running on a Storm cluster is called a *topology*. There are two sources of streams in a Storm topology: spouts and bolts. A *spout* is a producer of streams, and a *bolt* is both a consumer and a producer of streams. Each spout or bolt can have multiple threads called *tasks*.

In our implementation, the stream of training examples is not actually generated by an external source, but is instead simulated by a spout reading the training examples from a file and sending each example to the master as a tuple. The master, workers, and matrix stores are all implemented as bolts. (More specifically, the workers and matrix stores run as tasks in a bolt.) To receive the training examples, the master must subscribe to the stream emitted by the spout. As the master receives each example, it updates its bookkeeping data and forwards the example to the appropriate worker. Each worker saves the training examples in its local memory. Based on its bookkeeping data, the master periodically sends a tuple to a worker requesting that it process a block of the ratings matrix. The worker then requests the corresponding user and item blocks from the matrix stores; the worker already has the ratings block in its own memory. When the user finishes processing the block, it sends the user and item blocks back to the matrix stores and informs the master that it has finished. The master then updates its bookkeeping data by unlocking the corresponding block row and column. The matrix stores simply send blocks when they are requested and save blocks in memory when they are received; when a block is requested for the first time, the matrix store initializes each entry to a uniformly random value between 0 and 1. To communicate with each other, a worker and the master must each subscribe to a stream emitted by the other; likewise for the workers and the matrix stores. The spout, master, workers, and matrix stores are all packaged into a topology and submitted to a Storm cluster to run.

5 Experimentation

We experimented running both Batch DSGD as well Streaming DSGD on the Movie Lens [4] dataset, consisting of 1 million movie ratings from 6,000 users on 4,000 movies, in an Amazon EC2 [5] cluster environment. For each algorithm, we used a cluster of 10 machines to parallelize computation. We also normalized the data to have a mean of 0 and a standard deviation of 1. We ran each algorithm on a training set, specifying the number of iterations and observed the root mean squared error of the learned low-rank matrix factorization on a test set that was not used in training. For Batch DSGD the number of iterations specifies the number of times a full set of stratum covering the full ratings matrix was looped over. For Streaming DSGD the number of iterations specifies the number of times that a data point was allowed to be processed. For example, if we set the number of iterations to be 5, then as data streams in and blocks are chosen for computation, once a data point has been processed 5 times it is never used for processing again. In this manner, we can control the number of times that each data point is observed and terminate after all data points have streamed in and been processed the maximum number of times. For both methods we used a fixed penalty term of $\lambda = 0.1$ and a fixed SGD step size of $\tau = 0.1$. Figure 2 plots the total elapsed time in milliseconds$\times 10^5$ vs the root mean squared error of the test set for each method.

[4] http://www.grouplens.org/node/73
[5] http://aws.amazon.com/ec2/

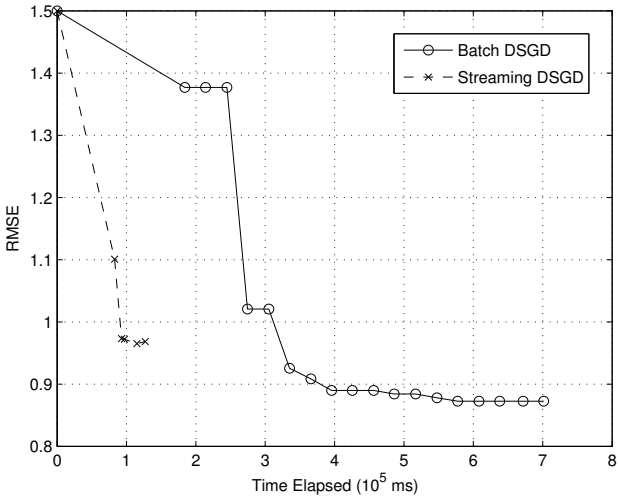

Figure 2: Plot of total time elapsed vs root mean squared error (RMSE)

As can be seen from this plot, Streaming DSGD is quicker to decrease its RMSE, however it levels out at a worse level than Batch DSGD. We believe that Streaming DSGD is able to converge quicker because there is not a bottleneck placed on the slowest worker. For example, in processing a stratum in Batch DSGD a new stratum cannot be processed until all blocks of the previous stratum have finished computation. If the distribution of ratings within each block is not uniform then it is likely that many blocks will finish processing early though no new computation can be performed until the slowest block has finished computation. In contrast, when a block finishes processing in Streaming DSGD, computation on a new block can begin immediately without waiting for a full stratum to complete. For this reason, Streaming DSGD was expected to perform faster than Batch DSGD. Please note however, that both implementations are dependent on their corresponding parallelization frameworks (Hadoop and Storm) and therefore such a comparison is of course prone to framework bias.

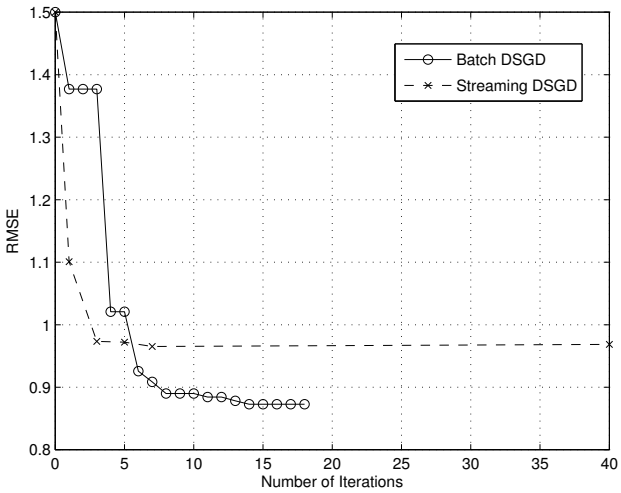

Figure 3: Plot of number of iterations (number of times each data point was looked at) vs root mean squared error (RMSE)

Figure 3 plots the number of iterations vs root mean squared error on the test set for both methods where number of iterations refers to the number of times that each data point was looked at. From this plot it can be seen that Streaming DSGD is actually able to outperform Batch DSGD under a low number

of iterations where as Batch DSGD eventually converges to a lower RMSE than Streaming DSGD. We believe that Streaming DSGD is able to outperform Batch DSGD in a low number of iterations because it employs more randomization than Batch DSGD. For example, the Batch DSGD algorithm we used chooses an initial stratum at random but then processes each consecutive stratum in order. We also didn't randomize ratings choices within each block process (which may have increased performance). On the streaming side, we shuffled the input ratings on initialization and chose which blocks to process completely at random (as opposed to choosing the next stratum in a deterministic ordering).

Neither implementation appeared to scale well as we moved into higher dimensional data. For example, we were not able to get either implementation working with the full Netflix dataset consisting of over 3 million ratings from 573 users on over 2 million movies. Batch SGD ran into heap space issues with buffering the full factor matrices on the reduce side. This could possibly have been fixed by only buffering the needed vectors as opposed to indexing the vectors by an array of the full size of the matrix, which is actually an inefficient use of space. For Streaming DSGD, using the full Netflix dataset caused Storm to become clogged with messages which exceeded the buffer size and caused processes to terminate. To get around this issue we introduced sleep times into each worker which in turn hurt performance of the method considerably. However, we believe these to be framework and implementation issues rather than algorithm issues and that both algorithms in their general form are expected to scale well with big data, especially since neither algorithm assumes the full ratings matrix to be held in memory.

6 Conclusion

We have extended the Batch DSGD algorithm from [4] to the case of streaming data and have even suggested that our method could handle batch data faster than the original DSGD method. We have further corroborated these claims with experimental results. In the future we would like to investigate further heuristics and new parallelization frameworks for implementing both methods. One such heuristic would be to cluster ratings data by additional features. For Batch SGD we would want to distribute ratings uniformly so that there were no weak links in a stratum which would in turn increase efficiency of processing stratums in parallel. For Streaming SGD we would want to cluster ratings into distinct blocks such that we had very highly concentrated blocks and then very low concentrated blocks (possibly empty blocks). Doing so would then allow us to choose blocks to process probabilistically based on a function of the number of data points in the block. This would help process the maximum number of data points on any worker process. As an example, consider movie ratings where the locations of users and the genres of movies were known. Then, we could cluster users of similar locations together (in terms of matrix index locality) as well as genres of movies (in terms of matrix index locality). The expectation is that users of similar locations will rate similar movies and movies of similar genres will get rated by the same types of people. Then, the expectation is that we would end up with both concentrated and sparse blocks but nothing in between. Then, by increasing the likelihood of choosing the larger blocks the method should be more efficient and decrease the overhead of message passing.

References

[1] Jacob Abernethy, Kevin Canini, John Langford, and Alex Simma. Online collaborative filtering. `http://www.cs.berkeley.edu/~kevin/research_files/OnlineCollaborativeFiltering.pdf`. Manuscript.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation—Volume 6*, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[3] Simon Funk. Netflix update: Try this at home. `http://sifter.org/~simon/journal/20061211.html`, 2006.

[4] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, pages 69–77, 2011.

[5] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[6] Jimmy Lin and Chris Dyer. *Data Intensive Text Processing with Map Reduce*. Morgan and Claypool Publishers, 2010.

[7] Nathan Marz. Storm source code. `https://github.com/nathanmarz/storm`.

[8] Nathan Marz. A storm is coming: more details and plans for release. `http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html`, 4 August 2011.

[9] Sebastian Schelter. Apache Mahout implementation of als-wr. `http://mahout.apache.org/`, 2011.

[10] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the Netflix prize. *Algorithmic Aspects in Information and Management*, 5034:337–348, 2008.