CME 323 PROJECT REPORT

"LARGE-SCALE MATRIX FACTORIZATION WITH DISTRIBUTED STOCHASTIC GRADIENT DESCENT: IMPLEMENTATION IN SPARK & TESTING ON NETFLIX MOVIE RECOMMENDATION"

Submitted to
Nolan Skochdopole & Andreas Santucci

BY

Wissam Baalbaki

Stanford University
ICME
CME 323

Spring 2016

Professor Reza Zadeh

ABSTRACT

In this project, we discuss ”Collaborative Filtering” as the most prominent approach for Recommender Systems. We analyze various ”Matrix Factorization” methods in the context of ”Distributed Computing”. Then, we provide our own implementation of the Distributed Stochastic Gradient Descent matrix factorization method ”DSGD” in Spark. DSGD was initially suggested by Rainer Gemulla, Peter J. Haas, Erik Nijkamp and Yannis Sismanis in their paper titled ”Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent”.

Using the ”Netflix Problem dataset”, we analyze the performance of DSGD in a Distributed Computing setting, compare it to the Alternating Least Squares Matrix Factorization Method, which is already implemented in Spark, and make recommendations.

Contents

Chapter 1

Introduction

1.1 Recommender Systems

Recommender systems are widely used on the Web for recommending products and services to users. Recommendations are context dependent (e.g. device, location, time) and can happen through different interfaces (e.g. mobile browser, tablet, viewport).

These systems serve two important functions:

1. They help users deal with the information overload by giving them recommendations of products, etc.
2. They help businesses make more profits, i.e., selling more products.

Some examples of Recommender Systems in our everyday life are:

- Movie recommendation (Netflix)
- Related product recommendation (Amazon)
- Web page ranking (Google)
- Social recommendation (Facebook)
- News content recommendation (Yahoo)
- Priority inbox spam filtering (Google)
- Online dating (OK Cupid) Computational Advertising (Yahoo)

While being common and studied for a long time, Recommender Systems still face major challenges:

1. Scalability: In a typical Recommender System problem, there might be millions of objects and hundreds of millions of users. Hence, any algorithm performing the recommendation should be able to work on such huge datasets.

2. Cold_Start: Recommender Systems posses a cold start issue. Both the users and the objects keep on changing so for any new user we wouldn't have any information about his preference for objects in the past and for any new object we wouldn't have any information about its appeal to users.

3. Imbalanced Dataset: The user activity/ item reviews are power law distributed. This means that we have most movies with very small number of reviews and very few movies with a huge number of reviews associated with them. From a user perspective, the number items reviewed is a small percentage of all items.

1.2 Collaborative Filtering

As it stands today, Collaborative Filtering is the most prominent approach to generate recommendations. It's used by large, commercial e-commerce sites and applicable in many domains (movies, books, DVDs,...).

The idea behind Collaborative Filtering is using the "Wisdom of Crowd" to recommend items. User input can be either implicit (either buy an object or not) or explicit (give a score to an object).

We define different Collaborative Filtering algorithms:

- Baseline Collaborative Filtering: Uses mean of all available ratings u, a bias for each item b_i (how better/worse is a specific item rated relative to other items) and a bias for each user b_u (how better/worse are the ratings of a specific user relative to other users). Then, the rating of item i by a specific user u would be: $r_{u,i} = u + b_i + b_u$ (a convex least squares problem).
- Memory-Based Collaborative Filtering: Recommends items to a specific user based the proximity of all items to the items the user liked in the past. Cosine similarity is one proximity measure that could be used.
- Matrix Factorization Collaborative Filtering: Views the combinations of items and users as a matrix and tries to decompose it into two smaller matrices. We will discuss Matrix Factorization in details in the following chapter.

Chapter 2

Matrix Factorization: Popular Models

Matrix factorization is a latent factor model. It assumes that there are some latent factors (can also be called: features, aspects) aspects Latent variables (also called features, aspects, or factors) that determine how a specific user rates a specific item. Thus, the problem shrinks down to finding those features, which would allow us to predict a rating with respect to a particular user/item combination. In trying to find the different features, we assume that they are less than both the number of users and the number of items since it would unreasonable to assume that each user is associated with a unique feature. This dimension reduction is one of the main advantages of Matrix Factorization compared to other Collaborative Filtering Techniques.

Additionally, Matrix factorization has had superior performance both in terms of recommendation quality and scalability. Scalability became a crucial measure as we entered the Big Data Era. A popular benchmark for testing new recommender systems is ”The Netflix Problem” and a Matrix Factorization method, namely Singular Value Decomposition (SVD), has won the Netflix Prize Contest.

2.1 Principal Component Analysis

Principal Component Analysis (PCA) is a powerful technique of dimensionality reduction and is a particular realization of the Matrix Factorization (MF) approach. PCA projects a dataset to a new coordinate system by determining the eigenvectors and eigenvalues of a matrix. It involves a calculation of a covariance matrix of a dataset to minimize the redundancy and maximize the variance. The covariance matrix is used to measure how much the dimensions vary from the mean with respect to each other. The covariance of two random variables (dimensions) is their tendency to vary together as:

$$cov(X,\,Y) = E[E[X]-X]\cdot E[E[Y]-Y]$$

where $E[X]$ and $E[Y]$ denote the expected value of X and Y respectively. For a sampled dataset, this can be explicitly written out.

$$cov(X,\ Y) = \sum_{i=1}^{N} \frac{(x_i - \bar{x})(y_i - \bar{y})}{N}$$

with $\bar{x} = mean(X)$ and $\mathrm{y} = mean(Y)$, where N is the dimension of the dataset. The covariance matrix is a matrix A with elements $A_{i,j} = cov(i,\ j)$. It centers the data by subtracting the mean of each sample vector.

Once the unit eigenvectors and the eigenvalues are calculated, the eigenvalues are sorted in descending order. This gives us the components in order of significance. The eigenvector with the highest eigenvalue is the most dominant principle component of the dataset (PC_1) . It expresses the most significant relationship between the data dimensions. Therefore, principal components are calculated by multiplying each row of the eigenvectors with the sorted eigenvalues.

The main limitation of PCA is it's inability to handle missing values (i.e. it would assume that the rating is 0 if the user hasn't rated an item. A modified version of PCA is Binary PCA, which finds components from data assuming Bernoulli distributions for the observations. Such probabilistic approach allows for straightforward treatment of missing values. This would allow us to deal better with the sparsity of Recommender Systems matrices.

2.2 Singular Value Decomposition

SVD is a well-known matrix factorization technique that factors an $m \times n$ matrix R into three matrices as the following:

$$R = U \cdot S \cdot V'$$

Where, U and V are two orthogonal matrices of size $m \times r$ and $n \times r$ respectively; r is the rank of the matrix R. S is a diagonal matrix of size $r \times r$ having all singular values of matrix R as its diagonal entries. All the entries of matrix S are positive and stored in decreasing order of their magnitude. The matrices obtained by performing SVD are particularly useful for our application because of the property that SVD provides the best lower rank approximations of the original matrix R, in terms of Frobenius norm. It is possible to reduce the $r \times r$ matrix S to have only k largest diagonal values to obtain a matrix $S_k, k < r$. If the matrices U and V are reduced accordingly, then the reconstructed matrix $R_k = U_k.S_k.V_k'$ is the closest rank-k matrix to R. In other words, R_k minimizes the Frobenius norm $||R\text{-}\ R||$ over all rank-k matrices.

SVD is used in recommender systems to perform two different tasks:

- To capture latent relationships between users and items that allow us to compute the predicted likeliness of a certain item by a user
- To produce a *low-dimensional* representation of the original user-item space and then compute neighborhood in the reduced space

Using the *low-dimensional* representation, a list of *top-N* item recommendations for users is generated.

Below is a description of the steps involved:

(a) Start with a user-item ratings matrix that is very sparse, we call this matrix R

(b) Remove Sparsity by assigning the average rating of an item to all missing ratings for that item

(c) Normalize ratings per user (i.e. subtract of user average from each rating so that the average rating of each user is 0)

(d) Get a normalized matrix R_{norm}. Essentially, $R_{norm} = R + NPR$, where NPR is the fill-in matrix that provides *naive non-personalized recommendation*

(e) Factor the matrix R_{norm} and obtain a *low-rank* approximation

(f) Use resultant matrices to compute the recommendation score for any user c and item p

The dimension of $U_k S_k^{1/2}$ is $m \times k$ and the dimension of $S_k^{1/2} V_k'$ is $k \times n$. To compute the prediction we simply calculate the dot-product of the c^{th} row of $U_k S_k^{1/2}$ and the p^{th} column of $S_k^{1/2} V_k'$ and add the user average back using the following:

$C_{P_{pred}} = \overline{C} + U_K . \sqrt{S_k}'(c) \cdot \sqrt{S_k} . V_{k'}(P)$.

Note that even though the R_{norm} matrix is dense, the special structure of the matrix NPR allows us to use sparse SVD algorithms whose complexity is almost linear to the number of non-zeros in the original matrix R. The optimal choice of the value k is critical to high-quality prediction generation. We are interested in a value of k that is large enough to capture all the important structures in the matrix yet small enough to avoid over-fitting errors. Usually, a good value of k is found by trying several different values.

SVD is able to handle large dataset, sparseness of rating matrix and scalability problem of CF algorithm efficiently. The prize winning method of the Netflix Prize Contest employed an adapted version of SVD

2.3 Alternating Least Squares

When it comes to using matrix factorization in Recommender Systems, our goal is to finding an $m \times r$ matrix W and an $r \times n$ matrix H such that $V \approx WH$ for a given $m \times n$ input matrix V, in the sense of minimizing a specified loss function $L(V,\ WH)$ computed over N training points.

Alternating Least Squares. In its standard form, the method of alternating least squares optimizes

$$L_{\mathrm{SL}} = \sum_{i,j} (V_{ij} - [WH]_{ij})^2$$

The method alternates between finding the best value for W given H, and finding the best value of H given W. This amounts to computing the least squares solutions to the following systems of linear equations

$$V_{i*} - W_{i*} H_n = 0,$$

$$V_{*j} - W_{n+1} H_{*j} = 0,$$

where the unknown variable is underlined. This specific form suggests that each row of W can be updated by accessing only the corresponding row in the data matrix V, while each column in H can be updated by accessing the corresponding column in V. This facilitates distributed processing; see below. The equations can be solved using a method of choice. We obtain

$$W_{n+1}^{\mathrm{T}} \leftarrow (H_n H_n^{\mathrm{T}})^{-1} H_n V^{\mathrm{T}},$$

$$H_{n+1} \leftarrow (W_{n+1}^{\mathrm{T}} W_{n+1})^{-1} W^{\mathrm{T}} V.$$

for the unregularized loss shown above. When an additional L_2 regularization term of form $\lambda(\|W\|_{\mathrm{F}}^2 + \|H\|_{\mathrm{F}}^2)$ is added, we obtain

$$\begin{aligned} W_{n+1}^{\mathrm{T}} &\leftarrow (H_n H_n^{\mathrm{T}} + \lambda I)^{-1} H_n V^{\mathrm{T}} \\ H_{n+1} &\leftarrow (W_{n+1}^{\mathrm{T}} W_{n+1} + \lambda I)^{-1} W^{\mathrm{T}} V \end{aligned}$$

Since the update term of H_{n+1} depends on W_{n+1}, the input matrix has to be processed twice to update both factor matrices.

In contrast to SVD, ALS does not produce an orthogonal factorization and it might get stuck in local minima. However, ALS can handle a wide range of variations for which SVD is not applicable, but which are important in practice. Examples include non-negativity constraints, sparsity constraints, weights and regularization. In general, ALS is applicable when the loss function is quadratic in both W and H.

In Spark MLlib, a distributed version of ALS is implemented for Recommender Systems. It breaks the matrices W and H 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 us 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.

2.4 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD), is an iterative optimization algorithm which has been shown, in a sequential setting, to be very effective for matrix factorization.

The goal of SGD is to find the value $\theta^* \in \Re^k (k \geq 1)$ that minimizes a given loss $L(\theta)$. The algorithm makes use of noisy observations $\hat{L}'(\theta)$ of $L'(\theta)$, the function's gradient with respect to θ. Starting with some initial value θ_0, SGD refines the parameter value by iterating the stochastic difference equation

$$\theta_{n+1} = \theta_{n^- \in n} \hat{L}'(\theta_n)$$

where n denotes the step number and $\{\in_n\}$ is a sequence of decreasing step sizes. Since $-L'(\theta_n)$ is the direction of steepest descent, (2) constitutes a noisy version of gradient descent.

Stochastic approximation theory can be used to show that, under certain regularity conditions, the noise in the gradient estimates "averages out" and SGD converges to the set of stationary points satisfying $L'(\theta) = 0$.

In order to apply SGD as a matrix factorization method, we set $\theta = (W,\ H)$ and decompose the loss L as in (1) for an appropriate training set Z and local loss function l. Denote by $L_z(\theta) = L_{ij}(\theta) = l(V_{ij},\ W_{i*},\ H_{*j})$ the local loss at position $z = (i,\ j)$. We have $L'(\theta) = \sum_z L'_z(\theta)$ by the sum rule for differentiation. DGD methods exploit the summation form of $L'(\theta)$ at each iteration by computing the local gradients $L'_z(\theta)$ in parallel and summing them. In contrast to this exact computation of the overall gradient, SGD obtains noisy gradient estimates by scaling up *just one* of the local gradients, i.e., $\hat{L}'(\theta) = NL'_z(\theta)$,where $N = |Z|$ and the training point z is chosen randomly from the training set. Algorithm 1 uses SGD to perform matrix factorization.

Algorithm 1 SGD for Matrix Factorization

```
Require: A training set $Z$, initial values $W_0$ and $H_0$
while not converged do $/ * step^* /$
Select a training point $(i,\ j) \in Z$ uniformly at random.
    $$W'_{i*} \leftarrow W_{i*} - \in_n N \frac{\partial}{\partial W_{i*}} l(V_{ij},\ W_{i*},\ H_{*j})$$
    $$H_{*j} \leftarrow H_{*j} - \in_n N \frac{\partial}{\partial H_{*j}} l(V_{ij},\ W_{i*},\ H_{*j})$$
    $$W_{i*} \leftarrow W'_{i*}$$
end while
```

Note that, after selecting a random training point $(i,\ j) \in Z$, all we have to update is W_{i*} and H_{*j}. We don't have to update the factors of the form $W_{i'*}$ for $i' \neq i$ or $H_{*j'}$ for $j' \neq j$. That's because we are representing the global loss as a summation of all local losses. In particular,

$$\frac{\partial}{\partial W_{i'k}} L_{ij}(W,\ H) = \begin{cases} 0 & \text{if } i \neq i' \\ \frac{\partial}{\partial W_{ik}} l(V_{ij},\ W_{i*},\ H_{*j}) & \text{otherwise} \end{cases}$$

and

$$\frac{\partial}{\partial H_{kj'}} L_{ij}(W,\ H) = \begin{cases} 0 & \text{if } j \neq j' \\ \frac{\partial}{\partial H_{kj}} l(V_{ij},\ W_{i*},\ H_{*j}) & \text{otherwise} \end{cases}$$

for $1 \leq k \leq r$.

SGD is considered an *online learning* algorithm as it averages multiple local losses rather than the exact loss at each step which we calculate in GD. This naturally raises the questions about its merits as an alternative. The problem with getting the exact losses using GD is basically its very high computation costs. Hence, using noisy estimates for the gradient rather than the exact gradient is much cheaper computationally and we would be able to go many updates in SGD in the same amount of time it would take us to make one GD update. 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. This behavior can be proven for some problems, and it has been observed in the case of large-scale matrix factorization. It's crucial to note that ALS has higher time complexity per iteration compared to SGD.

Chapter 3

Distributed Stochastic Gradient Descent

3.1 Overcoming Sequential Nature of SGD

The problem with SGD is that it is an inherently sequential algorithm. Even when we divide the matrix into blocks (called Stratified SGD ”SSGD”) and pick blocks to update at the same time, blocks’ updates can be dependent (i.e. we need the update in one block in order to update another block. A simple representation is shown below:

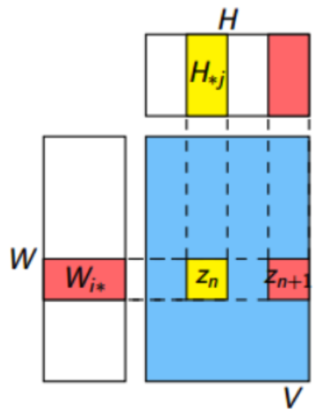


Figure 3.1: SGD Sequential Update

Clearly from the figure above, we can’t both z_n and z_{n+1} along with their corresponding factor matrices simultaneously because the updates are dependent. Both updates use W_{i*} and update it so z_n has to be updated before proceeding with updating z_{n+1}.

The key idea behind DSGD is to stratify(divide into blocks) the training set Z into a set $S = \{Z_1, \dots, Z_q\}$ of q strata so that each individual stratum $Z_s \subseteq Z$ can be processed in a distributed fashion. We do this by ensuring that each stratum is d-monomial”.

A stratum Z_s is d-monomial if it can be partitioned into d nonempty subsets $Z_s^1, Z_s^2, \dots, Z_s^d$ such that $i \neq i'$ and $j \neq j'$ whenever $(i, j) \in Z_s^{b_1}$ and $(i', j') \in Z_s^{b_2}$

with $b_1 \neq b_2$. A training matrix Z_s is d-monomial if it is constructed from a d-monomial stratum Z_s.

For example, z_n and z_{n+1} shown below can be updated at the same time:

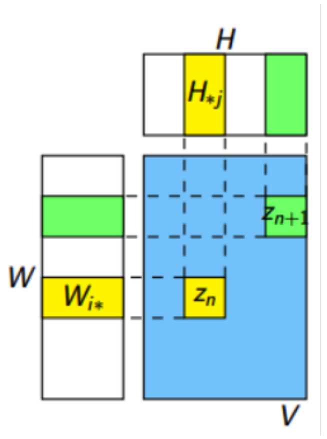


Figure 3.2: SSGD Parallel Update

The idea behind DSGD is to find such blocks that have no data in common and update them simulatenously.

The strata must cover the training set in that $\bigcup_{s=1}^{q} Z_s = Z$ and the parallelism parameter d is chosen to be greater than or equal to the number of available processing tasks.

We first randomly permute the rows and columns of Z, and then create $d \times d$ blocks of size $(m/d) \times (n/d)$ each; the factor matrices W and H are blocked conformingly. This procedure ensures that the expected number of training points in each of the blocks is the same, namely, N/d^2. Then, for a permutation $j_1, j_2, \ldots,\ j_d$ of 1, 2, . . . , d, we can define a stratum as $Z_s = Z^{1j_1} \cup Z^{2j_2} \cup \cdots \cup Z^{dj_d}$, where the substratum Z^{ij} denotes the set of training points that fall within block Z^{ij}. Thus, a stratum corresponds to a set of blocks; the figure below shows the set of possible strata when $d = 3$.

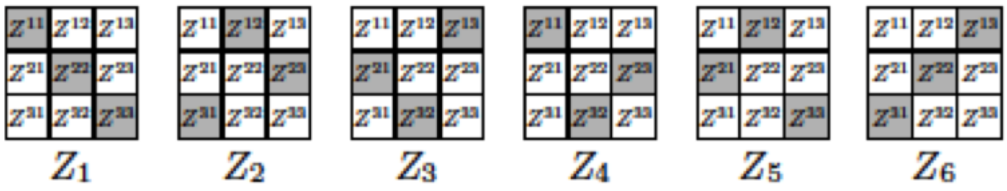

Z^{11} Z^{12} Z^{13}
Z^{21} Z^{22} Z^{23}
Z^{31} Z^{32} Z^{33}
Z_1
Z_2
Z_3
Z_4
Z_5
Z_6

Figure 3.3: Strata for a 3×3 blocking of training matrix V

In general, the set S of possible strata contains $d!$ elements, one for each possible permutation of 1, 2, . . . , d. We need 3 iterations in order to pass once *through all blocks of the original matrix. In each iteration, we select d disjoint strata (e.g. Z_1, Z_2 and Z_3 or Z_4, Z_5, Z_6) in figure 3.3*

Given a set of strata and associated weights $\{w_s\}$, the loss is decomposed into a weighted sum of per-stratum losses: $L(W,\ H) = \sum_{s=1}^{q} w_s L_s(W,\ H)$. Losses in each stratum can be expressed as:

$$L_s(W,\ H) = c_s \sum_{(i,j)\in Z_s} L_{ij}(W,\ H)$$

where c_s is a stratum-specific constant; see the discussion below. When running SGD on a stratum, we use the gradient estimate

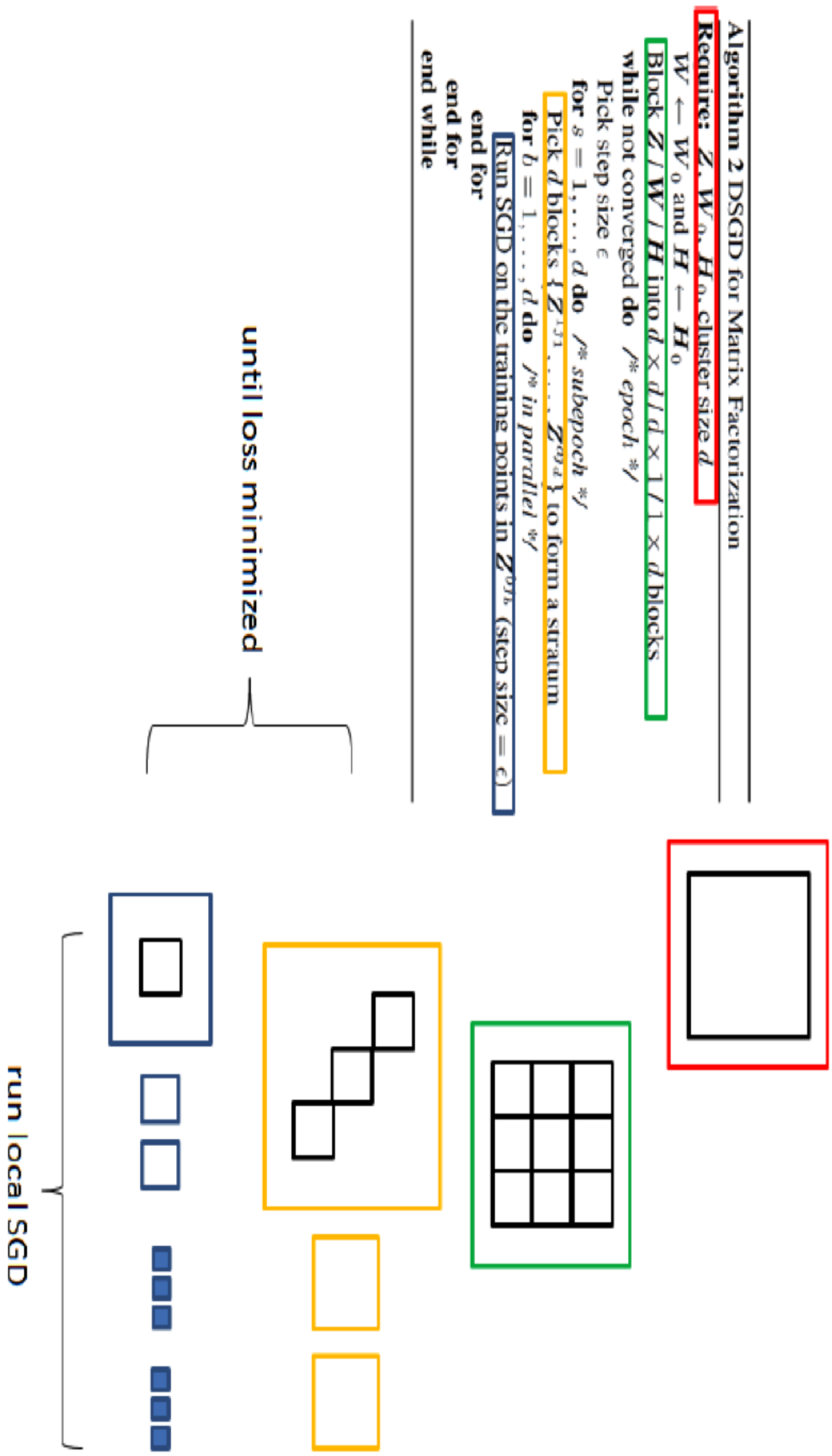
$$\hat{L}'_s(W,\ H) = N_s c_s L'_{ij}(W,\ H)$$

of $L'_s(W,\ H)$ in each step, i.e., we scale up the local loss of an individual training point by the size $N_s = |Z_s|$ of the stratum. Then any given loss function L of the form can be represented as a weighted sum over these strata by choosing w_s and c_s subject to $w_s c_s = 1$.

The individual steps in DSGD are grouped into subepochs, each of which amounts to processing one of the strata. The sequence of strata is chosen such that the underlying SSGD algorithm, and hence the DSGD factorization algorithm, is guaranteed to converge. Once a stratum ξ_k has been selected, we perform T_k SGD steps on Z_{ξ_k}; this is done in a parallel and distributed way using the SGD algorithm described in the previous section. DSGD is shown in the algorithm below, where we define an epoch as a sequence of d subepochs. Each epoch roughly corresponds to processing the entire training set once.

Algorithm 2 DSGD for Matrix Factorization

```
Require: $Z$, $W_0$, $H_0$, cluster size $d$
W← $W_0$ and $H \leftarrow H_0$
Block Z/W/H into $d \times d/d \times 1/1 \times d$ blocks
while not converged do $/ * epoch^* /$
Pick step size $\in$
for $s = 1, \dots, d$ do $/ * subepoch^* /$
Pick $d$ blocks $\{Z^{1j_1}, \dots, Z^{dj_d}\}$ to form a stratum
for $b = 1, \dots, d$ do $/ * in\ paralle\ l^* /$
Run SGD on the training points in $Z^{bj_b}$ (step size $= \in$)
end for end for end while
```



```
Algorithm 2 DSGD for Matrix Factorization
Require: $Z$, $W_0$, $H_0$, cluster size $d$
  $W \leftarrow W_0$ and $H \leftarrow H_0$
  Block $Z$ / $W$ / $H$ into $d \times d$ / $d \times 1$ / $1 \times d$ blocks
  while not converged do  /* epoch */
    Pick step size $\epsilon$
    for $s = 1, \ldots, d$ do  /* subepoch */
      Pick $d$ blocks $\{Z^{1j_1}, \ldots, Z^{dj_d}\}$ to form a stratum
      for $b = 1, \ldots, d$ do  /* in parallel */
        Run SGD on the training points in $Z^{bj_b}$ (step size = $\epsilon$)
      end for
    end for
  end while
```

Figure 3.4: DSGD Algorithm Step-by-Step

3.2 Implementation in PySpark

We present our code in PySpark below:

```


"""This script runs DSGD on matrix factorization using Spark"""
import os
import sys
import numpy as np
from numpy import linalg
import multiprocessing
import csv
from operator import itemgetter
from scipy import sparse



SPARK_HOME = "/Users/baalbaki/Documents/Stanford/CME323/Project/spark" #
    SPARK Path
PYTHONPATH=SPARK_HOME/python/+ PYTHONPATH
PYTHONPATH=SPARK_HOME/python/lib/py4j-0.8.2.1-src.zip+PYTHONPATH
os.environ["SPARK_HOME"] = SPARK_HOME
os.environ["SPARK_LOCAL_IP"] = "127.0.0.1" # Setting up Local IP
sys.path.append(SPARK_HOME + "/python")
from pyspark import SparkContext, SparkConf

def main(numberOfFactors, numberOfWorkers, iterations, modelBeta,
    modelLambda, inputPathOfV, outputPathOfW, outputPathOfH):
    print 'Algorithm in Progress'
    numberOfWorkers = int(numberOfWorkers)
    iterations = int(iterations)
    numberOfFactors = int(numberOfFactors)
    modelBeta = float(modelBeta)
    modelLambda = float(modelLambda)
    conf = SparkConf().setAppName('CME323Project').setMaster('local')
    #sc = SparkContext(conf=conf) #No need to define sc again
    if os.path.isdir(inputPathOfV):
        p = multiprocessing.Pool()
        allPaths = [os.path.join(inputPathOfV, f) for f in os.listdir(
            inputPathOfV)]
        res = p.map(rawFileFormat, allPaths)
        p.close()
        flatResults = [item for sublist in res for item in sublist]
        filename = open('output.txt','w')
        filename.write ('\n'.join(flatResults))
        data = sc.parallelize(flatResults)
    elif os.path.isfile(inputPathOfV):
        data = sc.textFile(inputPathOfV)
    else:
        raise Exception("Input File Path is invalid")

    #Formatting V
    splitRDD = data.map(lambda x: x.split(','))
    VRDD = splitRDD.map(lambda x: map(lambda y: int(y),x)).sortBy(lambda x:
         (x[0],x[1]))



    ##Initializing variables
    t = 100
```

```python
    numberOfCounts = 0
    currentIteration = 0
    isError = False
    reconstructionError = []


    if isError:
        WVectorRDD = splitRDD.map(lambda x: int(x[0])).distinct().sortBy(lambda x: x).map(lambda x: tuple([x,np.random.rand(1,numberOfFactors).astype(np.float32)]))
        HVectorRDD = splitRDD.map(lambda x: int(x[1])).distinct().sortBy(lambda x: x).map(lambda x: tuple([x,np.random.rand(numberOfFactors,1).astype(np.float32)]))
        V, selection = loadMatrixSparse(inputPathOfV)
        while currentIteration != iterations:
            for stratum in xrange(0,numberOfWorkers-1):
                keyedV = VRDD.keyBy(lambda x: x[0]%numberOfWorkers).partitionBy(numberOfWorkers)
                partitionedV = keyedV.filter(lambda x: (x[1][1]+stratum)%numberOfWorkers==x[0])
                keyedH = HVectorRDD.keyBy(lambda x: (x[0]+stratum)%numberOfWorkers).partitionBy(numberOfWorkers)
                keyedW = WVectorRDD.keyBy(lambda x: x[0]%numberOfWorkers).partitionBy(numberOfWorkers)
                RDDsCombined = partitionedV.groupWith(keyedH, keyedW)
                outputRDD = RDDsCombined.mapPartitions(lambda x: mapLossNZSL(x, numberOfFactors)).reduceByKey(lambda x,y: x+y)
                WVectorRDD = outputRDD.filter(lambda x: x[0]=='W').flatMap(lambda x: x[1])
                HVectorRDD = outputRDD.filter(lambda x: x[0]=='H').flatMap(lambda x: x[1])
                outputW = WVectorRDD.collect()
                outputW.sort()
                outputH = HVectorRDD.collect()
                outputH.sort()


                W = outputW[0][1]
                temporaryCount = 1
                for WIndex in xrange(2,outputW[-1][0]+1):
                    if outputW[temporaryCount][0] == WIndex:
                        W = np.vstack([W, outputW[temporaryCount][1]])
                        temporaryCount += 1
                    else:
                        W = np.vstack([W, np.zeros((1,numberOfFactors))])

                H = outputH[0][1]
                temporaryCount = 1
                for HIndex in xrange(2,outputH[-1][0]+1):
                    if outputH[temporaryCount][0] == HIndex:
                        H = np.hstack([H, outputH[temporaryCount][1]])
                        temporaryCount += 1
                    else:
                        H = np.hstack([H, np.zeros((numberOfFactors,1))])


                error = calcError(V,W,H,selection)
                reconstructionError.append(error)
                print "Reconstruction error:", error
                currentIteration += 1
                if currentIteration == iterations:
                    break
    else:
        WVectorRDD = splitRDD.map(lambda x: tuple([int(x[0]),1])).reduceByKey(lambda x,y : x+y).map(lambda x: tuple([x[0],tuple([x[1],np.random.rand(1, numberOfFactors).astype(np.float32)])]))
        HVectorRDD = splitRDD.map(lambda x: tuple([int(x[1]),1])).reduceByKey(lambda x,y : x+y).map(lambda x: tuple([x[0],tuple([x[1],np.random.rand(numberOfFactors,1).astype(np.float32)])]))
        while currentIteration != iterations:
            for stratum in xrange(0,numberOfWorkers-1):
                keyedV = VRDD.keyBy(lambda x: x[0]%numberOfWorkers).partitionBy(numberOfWorkers)
                partitionedV = keyedV.filter(lambda x: (x[1][1]+stratum)%numberOfWorkers==x[0])
                keyedH = HVectorRDD.keyBy(lambda x: (x[0]+stratum)%numberOfWorkers).partitionBy(numberOfWorkers)
                keyedW = WVectorRDD.keyBy(lambda x: x[0]%numberOfWorkers).partitionBy(numberOfWorkers)
                RDDsCombined = partitionedV.groupWith(keyedH, keyedW)
                outputRDD = RDDsCombined.mapPartitions(lambda x: mapLoss(x, modelLambda, numberOfCounts, t, modelBeta, numberOfFactors)).reduceByKey(lambda x,y: x+y)
                WVectorRDD = outputRDD.filter(lambda x: x[0]=='W').flatMap(lambda x: x[1])
                HVectorRDD = outputRDD.filter(lambda x: x[0]=='H').flatMap(lambda x: x[1])
                numberOfCounts = (outputRDD.filter(lambda x: x[0]=='N').collect())[0][1]
                currentIteration += 1
                if currentIteration == iterations:
                    break

    if isError:
        HFile = open(outputPathOfH, 'h')
        WFile = open(outputPathOfW, 'w')
        NZSLPrintCsv(outputW, WFile, outputH, HFile, numberOfFactors)
        print reconstructionError
    else:
        outputW = WVectorRDD.collect()
        outputW.sort()
        WFile = open(outputPathOfW, 'w')
        outputH = HVectorRDD.collect()
        outputH.sort()
        HFile = open(outputPathOfH, 'h')
        W = outputW[0][1][1]
        temporaryCount = 1
        for WIndex in xrange(2,outputW[-1][0]+1):
            if outputW[temporaryCount][0] == WIndex:
                W = np.vstack([W, outputW[temporaryCount][1][1]])
                temporaryCount += 1
            else:
                W = np.vstack([W, np.zeros((1,numberOfFactors))])
        H = outputH[0][1][1]
        temporaryCount = 1
        for HIndex in xrange(2,outputH[-1][0]+1):
            if outputH[temporaryCount][0] == HIndex:
                H = np.hstack([H, outputH[temporaryCount][1][1]])
                temporaryCount += 1
            else:
                H = np.hstack([H, np.zeros((numberOfFactors,1))])
        np.savetxt(outputPathOfW, W, delimiter=",")
        np.savetxt(outputPathOfH, H, delimiter=",")

def LoadMatrix(csvfile):
    data = np.genfromtxt(csvfile, delimiter=',')
    return np.matrix(data)

def PrintCsv(outputW, WFile, outputH, HFile, numberOfFactors):
    temporaryCount = 0
    for WIndex in xrange(1,outputW[-1][0]+1):
        if outputW[temporaryCount][0] == WIndex:
            WFile.write(','.join(['%.5f' % num for num in (outputW[
                temporaryCount][1][1]).transpose()])+'\n')
            temporaryCount += 1
        else:
            WFile.write(','.join(['0']*numberOfFactors)+'\n')

    temporaryCount = 0
    for HIndex in xrange(1,outputH[-1][0]+1):
        if outputH[temporaryCount][0] == HIndex:
            HFile.write(','.join(['%.5f' % num for num in outputH[
                temporaryCount][1][1]])+'\n')
            temporaryCount += 1
        else:
            HFile.write(','.join(['0']*numberOfFactors)+'\n')
    WFile.close()
    HFile.close()

def NZSLPrintCsv(outputW, WFile, outputH, HFile, numberOfFactors):
    temporaryCount = 0
    for WIndex in xrange(1,outputW[-1][0]+1):
        if outputW[temporaryCount][0] == WIndex:
            WFile.write(','.join(['%.5f' % num for num in (outputW[
                temporaryCount][1]).transpose()])+'\n')
            temporaryCount += 1
        else:
            WFile.write(','.join(['0']*numberOfFactors)+'\n')

    temporaryCount = 0
    for HIndex in xrange(1,outputH[-1][0]+1):
        if outputH[temporaryCount][0] == HIndex:
            HFile.write(','.join(['%.5f' % num for num in (outputH[
                temporaryCount][1])])+'\n')
            temporaryCount += 1
        else:
            HFile.write(','.join(['0']*numberOfFactors)+'\n')
    WFile.close()
    HFile.close()
    return True

def rawFileFormat(filePath):
    output = []
    file = open(filePath)
    movie = (file.readline())[:-2]
    for line in file:
        ratings = line.split(",")
        output.append(ratings[0]+','+movie+','+ratings[1])
    return output

def mapLoss(keyedIter, modelLambda, numberOfCounts, t, modelBeta,
    numberOfFactors):
        iterableList = (keyedIter.next())[1]
        iterableV = iterableList[0]
        iterableH = iterableList[1]
        iterableW = iterableList[2]

        WDictionary = {}
        HDictionary = {}

        newWDictionary = {}
        newHDictionary = {}

        for elementOfH in iterableH:
            HDictionary[elementOfH[0]] = elementOfH[1]

        for elementOfW in iterableW:
            WDictionary[elementOfW[0]] = elementOfW[1]

        for elementOfV in iterableV:
            (i,j,rating) = elementOfV
            eps = np.power(t + numberOfCounts, -modelBeta)
            if j in HDictionary:
                inputH = HDictionary[j]
            else:
                HDictionary[j] = tuple([j,np.random.rand(numberOfFactors,1).astype(np.float32)])
                inputH = HDictionary[j]
            if i in WDictionary:
                inputW = WDictionary[i]
            else:
                WDictionary[i] = tuple([i,np.random.rand(1,numberOfFactors).astype(np.float32)])
                inputW = WDictionary[i]
            (WNew, HNew) = L2Loss(rating, inputH, inputW, modelLambda, eps)
            numberOfCounts += 1
            newWDictionary[i] = WNew
            newHDictionary[j] = HNew


        return (tuple(['W',newWDictionary.items()]), tuple(['H', newHDictionary.items()]), tuple(['N', numberOfCounts]))

def L2Loss(rating, elementOfH, elementOfW, modelLambda, eps):
    (NH, arrayOfH) = elementOfH
    (WN, arrayOfW) = elementOfW

    WOld = arrayOfW.copy()
    HOld = arrayOfH.copy()


    Grad = -2*(rating-np.asscalar(WOld.dot(HOld)))

    arrayOfW = np.add(WOld, np.multiply (eps, np.multiply(HOld.transpose(), Grad) + np.multiply(2*modelLambda/WN, WOld)))
    arrayOfH = np.add(HOld, np.multiply(eps, np.multiply(WOld.transpose(), Grad) + np.multiply(2*modelLambda/NH, HOld)))

    return (tuple([WN, arrayOfW]), tuple([NH, arrayOfH]))

def mapLossNZSL(keyedIter, numberOfFactors):
        iterableList = (keyedIter.next())[1]
        iterableV = iterableList[0]
        iterableH = iterableList[1]
        iterableW = iterableList[2]

        WDictionary = {}
        HDictionary = {}

        newWDictionary = {}
        newHDictionary = {}

        for elementOfH in iterableH:
            HDictionary[elementOfH[0]] = elementOfH[1]

        for elementOfW in iterableW:
            WDictionary[elementOfW[0]] = elementOfW[1]

        for elementOfV in iterableV:
            (i,j,rating) = elementOfV
            if j in HDictionary:
                inputH = HDictionary[j]
            else:
                HDictionary[j] = np.random.rand(numberOfFactors,1).astype(
                    np.float32)
                inputH = HDictionary[j]
            if i in WDictionary:
                inputW = WDictionary[i]
            else:
                WDictionary[i] = np.random.rand(1,numberOfFactors).astype(
                    np.float32)
                inputW = WDictionary[i]
            (WNew, HNew) = NZSLLoss(rating, inputH, inputW)
            newWDictionary[i] = WNew
            newHDictionary[j] = HNew

        return (tuple(['W',newWDictionary.items()]), tuple(['H',
            newHDictionary.items()]))

def NZSLLoss(rating, arrayOfH, arrayOfW):
    WOld = arrayOfW.copy()
    HOld = arrayOfH.copy()
    Grad = -2*(rating-np.asscalar(WOld.dot(HOld)))

    arrayOfW = np.add(WOld, np.multiply(HOld.transpose(), Grad))
    arrayOfH = np.add(HOld, np.multiply(WOld.transpose(), Grad))

    return (arrayOfW, arrayOfH)

def loadMatrixSparse(csvfile):
        value = []
        rows = []
        columns = []
        selection = []
        file = open(csvfile)
        readerFile = csv.reader(file)
        for line in readerFile:
                rows.append( int(line[0])-1 )
                columns.append( int(line[1])-1 )
                value.append( int(line[2]) )
                selection.append( (int(line[0])-1, int(line[1])-1) )
        return sparse.csr_matrix( (value, (rows, columns)) ), selection

def calcError(V, W, H, selection):
        print
        difference = V-W.dot(H)
        err = 0
        for rows, columns in selection:
                err += difference[rows, columns]*difference[rows, columns]
        return err/len(selection)

if __name__ == "__main__":

    #main(10,2,1,0.8,1.0,'test.txt','w.csv','h.csv')
```

Chapter 4

Results

4.1 Analysis

In this section, we analyze our the performance of DSGD as a matrix factorization method for Recommender Systems. We look at different aspects:

4.1.1 Convergence

As noted previously, DSGD is ***guaranteed to converge****. In our tests, DSGD and ALS converged in the same manner but DSGD reported higher Risk Square Mean Error (RMSE). However,* ***the difference in RMSE between them wasn't significant*** *but it's important to note that both methods converge faster than other methods and achieve lower RMSE.*

4.1.2 Communication Costs

In DSGD, the d-monomial strata representation has reduced communication costs. We have a ***block to block communication*** *between epochs. Each machine will have to send data to one single machine and receive data from one single machine.*

4.1.3 Shuffle Size

When executing DSGD on d nodes in a shared-nothing environment such as SPARK, the input matrix is only distributed once. Then, ***the only data that are transmitted between nodes during subsequent processing are (small) blocks of factor matrices****. In this DSGD implementation, node i stores blocks* W^i, Z^{i1}, Z^{i2}, $\ldots Z^{id}$ *for* $1 \leq i \leq d$*; thus only matrices* $H^1, H^2, \ldots, H^d$ *need be transmitted. (If the* W^i *matrices are smaller, then we transmit these instead). Note that DSGD only shuffles strata and blocks where as SGD shuffles all data.*

4.1.4 Scalability

- ***Input Size***: *We tested the DSGD algorithm in SPARK on a sample of Netflix Problem dataset and we used different size of data and different number of available ratings (i.e. different sparsity levels).*

Rank	*Wall Clock Time Per Epoch (s)*
50	120
100	125
200	135

  DSGD ***scaled very well in terms of matrix dimensions and also number of values in a matrix***.

- ***Number of Machines***: *We also tested the same dataset on different number of cores on a local machine (1 core, 2 cores, 4 cores). We also ran the ALS method already implemented in MLlib as a reference for comparison. Table Below shows results.*

Wall Clock Time		
Number of Cores	*Our DSGD*	*MLlib ALS*
1	X	X
2	$0.50X$	$0.51X$
4	$0.26X$	$0.25X$

  DSGD ***scaled almost linearly*** *when the number of cores was small.*

  Since DSGD's majority of the time is spent on communication costs,we suspected that the scalability might not be as efficient if the number of cores become too large so that the size of each stratum block is relatively small. To verify this intuition, we tested the dataset on AWS and used multiple cores (8 cores, 16 cores, 32 cores, 46 cores). The table below shows the results:

of Cores	*Wall Clock Time Per Epoch*
8	X
16	$0.52X$
32	$0.27X$
64	$0.24X$

  As expected, ***the scalability of DSGD got much worse when the number of cores increased relative to the input size***.

4.2 Conclusion & Future Scope

In practice, e-commerce sites like amazon.com experience tremendous amount of user visits per day. Recommending items to these large number of users in real-time requires the underlying recommendation engine to be highly scalable. Considering that the majority of time spent by SPARK ALS is on calculations rather than communication, SPARK ALS scales better than DSGD in practice. That is confirmed by the extensive use of parallel ALS in practice.

Regarding convergence, DSGD is proven to converge while there is no theory that SPARK ALS must converge; however, feedback reported from practical applications have confirmed that SPARK ALS converges faster than DSGD.

Since training data is sparse and we stratify the training set using data-independent blocking, a block Z^b may contain no training points; in this case we cannot execute SGD on the block, so the corresponding factors simply remain at their initial values.

Hence, we would like to examine other ways to stratify which could be data dependent so as to keep the size of available rating in each block similar for example. That should reduce our convergence time.

It's very important to note that SPARK is much more efficient that MapReduce for implementing DSGD as it operates on data in memory and it doesn't have to write to disk after each iteration. However, We recommend MLlib ALS as the better Matrix Factorization approach for Recommender Systems.

References

[1] Rainer Gemulla, Peter J. Haas, Erik Nijkamp, Yannis Sismanis; ”Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent”*; Max-Planck-Institut fur Informatik, IBM Almaden Research Center*

[2] Lslzo Kozma, Alexander Ilin, Tapani Raiko; ”Binary Principal Component Analysis in the Netflix Collaborative Filtering Task”*; Helsinki University of Technology*

[3] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, John T. Riedl; ”Application of Dimensionality Reduction in Recommender System – A Case Study”*; University of Minnesota, GroupLens Research Group*

[4] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, Chih-Jen Lin; ”A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems”*; National Taiwan University*

[5] Fanglin Li, Bin Wu, Liutong Xu, Chuan Shi, Jing Shi; ”A Fast Distributed Stochastic Gradient Descent Algorithm for Matrix Factorization”*; Beijing University of Posts and Telecommunications*

[6] Dheeraj kumar Bokde, Sheetal Girase, Debajyoti Mukhopadhyay; ”Role of Matrix Factorization Model in Collaborative Filtering Algorithm: A Survey”*; Maharashtra Institute of Technology*

[7] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, Rong Pan; ”Large-scale Parallel Collaborative Filtering for the Netflix Prize”*; HP Labs, Palo Alto*