DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF GEORGIA

CSCI6900 Assignment 4: SGD for Matrix Factorization on Spark

DUE: Monday, November 9 by 11:59:59pm

Out October 19, 2015

1 OVERVIEW

In this assignment, we will implement Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent (DSGD-MF) in Spark. The paper sets forth a solution for matrix factorization using minimization of sum of local losses. The solution involves dividing the matrix into strata for each iteration and performing sequential stochastic gradient descent within each stratum in parallel. The two losses considered are the plain non-zero square loss and the non-zero square loss with l_2 regularization:

$$L_{ij} = l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = (\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})^2 \tag{1.1}$$

$$L_{NZSL} = \sum_{(i,j)\in\mathbf{Z}} L_{ij} \tag{1.2}$$

$$L_2 = L_{NZSL} + \lambda\{\|\mathbf{W}\|_F^2 + \|\mathbf{H}\|_F^2\} \tag{1.3}$$

(if you are not familiar with regularization, look at the linked Wikipedia article; it is primarily a way to ensure that iterative algorithms do not wildly overshoot or overfit, but rather enforce smaller, smoother updates to the parameters being learned)

DSGD-MF is a fully distributed algorithm, i.e. both the data matrix **V** and factor matrices **W** and **H** can be carefully split and distributed to multiple workers for parallel computation without communication costs between the workers. Hence, it is a good match for implementation in a distributed in-memory data processing system like Spark. We outline the sequential algorithm and describe the steps needed to make it ready for distributed execution.

We've covered similar material in this class previously. In particular, see Lecture 9 (Classification and Regression; slides 36, 37), Lecture 14 (Randomized Algorithms; slides 4-8), and Lecture 16 (Factorbird).

2 SGD for Matrix Factorization

Please read through the linked paper on DSGD-MF as you are reading through this assignment.

Algorithm 1 provides an overview of the first part. In it, we select a single datapoint and update the corresponding row of **W** and column of **H** in the direction of negative gradient. The gradients for L_{NZSL} (Eq. 1.2) loss are given as follows:

$$\frac{\partial}{\partial \mathbf{W}_{i*}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})\mathbf{H}_{*j} \tag{2.1}$$

$$\frac{\partial}{\partial \mathbf{H}_{*j}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})\mathbf{W}_{i*}^T \tag{2.2}$$

The gradients for L_2 loss (Eq. 1.3) are given as follows:

$$\frac{\partial}{\partial \mathbf{W}_{i*}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})\mathbf{H}_{*j} + 2\frac{\lambda}{N_{i*}}(\mathbf{W}_{i*})^T \tag{2.3}$$

$$\frac{\partial}{\partial \mathbf{H}_{*j}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})\mathbf{W}_{i*}^T + 2\frac{\lambda}{N_{*j}}\mathbf{H}_{*j} \tag{2.4}$$

They're identical to the gradient equations 2.1 and 2.2 above, but with additive regularization terms tacked onto the ends. These are the updates you'll compute in each iteration of SGD. Note, however, that the calculation of the gradient L_{ij} and its use in updating **W** and **H** is dependent on the entry $\mathbf{V}_{ij}$, and therefore the i^{th} row of **W** ($\mathbf{W}_{i*}$) and the j^{th} column of **H** ($\mathbf{H}_{*j}$). Therefore, the algorithm as written is not trivially parallelizable: updating a single element $\mathbf{V}_{ij}$ requires somehow "locking" the corresponding rows and columns of **W** and **H** respectively from being updated by other workers.

3 Stratified SGD for Matrix Factorization

A pair of elements of a matrix given as (i, j) and (i', j') is interchangeable if $i \neq i'$ and $j \neq j'$. See Fig. 3.1 for an example. If two such elements are interchangeable, then the stochastic gradient descent updates involving $\{\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}\}$ and $\{\mathbf{V}_{i'j'}, \mathbf{W}_{i'*}, \mathbf{H}_{*j'}\}$ do not depend on each

Algorithm 1 SGD for Matrix Factorization

```
Require: Training indices $\mathbf{Z}$ ($N = |\mathbf{Z}|$), training data $\mathbf{V}$, randomly initialized $\mathbf{W}_0$ and $\mathbf{H}_0$
  while not converged do
    Select a training point $(i, j) \in \mathbf{Z}$ uniformly at random
    $\mathbf{W}'_{i*} \leftarrow \mathbf{W}_{i*} - \epsilon_n N \frac{\partial}{\partial \mathbf{W}_{i*}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j})$
    $\mathbf{H}'_{*j} \leftarrow \mathbf{H}_{*j} - \epsilon_n N \frac{\partial}{\partial \mathbf{H}_{*j}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j})$
    $\mathbf{W}_{i*} \leftarrow \mathbf{W}'_{i*}$
    $\mathbf{H}_{*j} \leftarrow \mathbf{H}'_{*j}$
  end while
```

other in any way and can be performed in parallel.

A set of elements of a matrix is interchangeable if any pair of elements in the set is interchangeable. The stochastic gradient descent updates involving the local losses of a set of interchangeable elements are also parallel due to the fact that they depend on disjoint parts of the data matrix **V** and the factor matrices **W** and **H**.

We can generalize the notion of interchangeability from elements of a matrix to blocks of the matrix. Consider I and I' are sets of row indices of **V**. Similarly, J and J' are sets of column indices of **V**. Matrix blocks IJ and $I'J'$ are interchangeable if $I \cap I' = \emptyset$ and $J \cap J' = \emptyset$, where IJ denotes the cartesian product of sets I and J, and $\mathbf{V}_{IJ}$ denotes a matrix block with some abuse and inelegance of notation. Similar to the case of element exchangeability, the stochastic gradient descent updates involving $\{\mathbf{V}_{IJ}, \mathbf{W}_{I*}, \mathbf{H}_{*J}\}$ and $\{\mathbf{V}_{I'J'}, \mathbf{W}_{I'*}, \mathbf{H}_{*J'}\}$ do not depend on each other in any way and can be performed in parallel.

As in the case of element interchangeability, block interchangeability generalizes from a pair of matrix blocks to a set of matrix blocks. A set of matrix blocks is said to be interchangeable if any pair of those matrix blocks is interchangeable. Stochastic gradient descent updates involving interchangeable matrix blocks can be parallelized since they depend on disjoint parts of the data matrix **V** and the factor matrices **W** and **H**.

Given a matrix block $\mathbf{V}_{IJ}$ and the coupled parameter blocks $\mathbf{W}_{I*}$ and $\mathbf{H}_{*J}$, we can perform sequential stochastic gradient descent for this disjoint part of the matrix and its parameters without having to worry about any parallel updates that might affect $\mathbf{W}_{I*}$ and $\mathbf{H}_{*J}$. Thus, we have established a concurrent model for stochastic gradient descent updates for matrix factorization. Interchangeable matrix blocks are also called *strata*, and hence this concurrent algorithm of performing SGD is also called Stratified Stochastic Gradient Descent (SSGD).

4 Distributed SGD for Matrix Factorization (DSGD-MF)

Having established concurrency in SSGD-MF, here we describe how to parallelize SSGD to yield Distributed Stochastic Gradient Descent algorithm for matrix factorization (DSGD-MF).

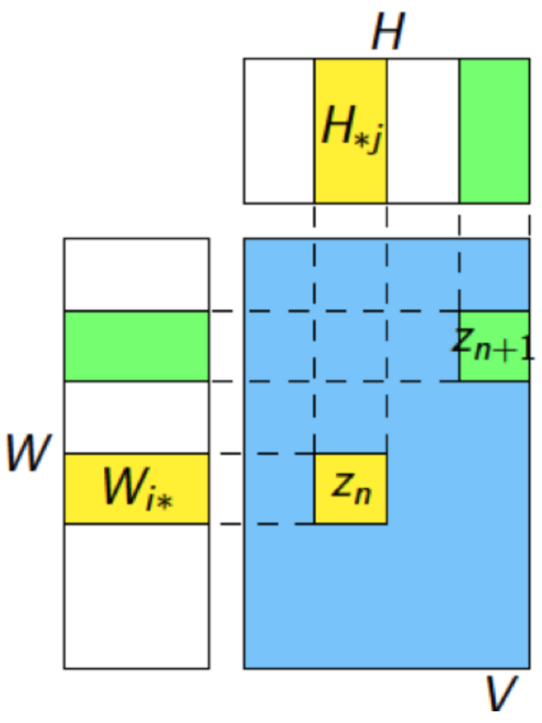

Figure 3.1: Element interchangeability in **V**.

Algorithm 2 provides the pseudocode. As long as the parameter estimates **W** and **H** have not converged, we choose a set of strata, which creates disjoint blocks of **V**, **W** and **H** that can be handed over to workers for performing sequential stochastic gradient descent. An example of such a stratification is shown in Fig. 5.1. Each worker performs SGD updates for the block of parameters it has been handed and returns the updated parameter blocks. For convenience, during iteration i, we will allow the n^{th} worker to perform m_{ni} updates, where m_{ni} is the number of non-zero entries in the data matrix block $\mathbf{V}_{I_nJ_n}$ handed to that worker during iteration i. The worker will sequentially go through all the non-zero entries in its block $\mathbf{V}_{I_nJ_n}$ and perform an SGD update for each entry as explained in SGD-MF earlier.

5 Data and Parameters

We will be using a subsample of the Netflix dataset for the experimental evaluation. This is a very popular recommendation dataset that has ratings of movies by Netflix users. The subsampled dataset is available at `http://ridcully.cs.uga.edu/assignment4/nf_subsample.csv`. You will perform your experimental evaluations on the subsampled Netflix dataset. It is in the triples format:

```
<user_1>,<movie_1>,<rating_11>
...
```

Algorithm 2 Distributed SGD for Matrix Factorization

```
Require: Training indices $\mathbf{Z}$, training data $\mathbf{V}$, randomly initialized $\mathbf{W}_0$ and $\mathbf{H}_0$, the number of
  workers $B$, the number of factors $F$
  while not converged do
    Select a stratum $S = (I_1, J_1), (I_2, J_2), ..., (I_B, J_B)$ of $B$ blocks
    {parallel for loop; what Spark operation would this be?}
    for $b \in 1...B$ do
      Get block of data matrix $\mathbf{V}_{I_b J_b}$
      Get blocks of parameter matrices $\mathbf{W}_{I_b *}$ and $\mathbf{H}_{* J_b}$
      Perform $m_b$ SGD updates for parameters $\mathbf{W}_{I_b *}$ and $\mathbf{H}_{* J_b}$
    end for
    Collect the updated parameter blocks from all workers and update $\mathbf{W}$ and $\mathbf{H}$
  end while
```

```
<user_i>,<movie_j>,<rating_ij>
...
<user_M>,<movie_N>,<rating_ij>
```

Here, `user_i` is an integer ID for the i^{th} user, `movie_j` is an integer ID for the j^{th} movie, and `rating_ij` is the rating given by `user_i` to `movie_j`.

For example, the content of the input file to your script would look like:

```
1,3114,4
1,608,4
1,1246,4
2,1357,5
2,3068,4
2,1537,4
...
6040,562,5
6040,1096,4
6040,1097,4
```

We will explore various choices for the number of iterations I, the number of workers B, the number of factors F, and the inner SGD step size ϵ_n. The SGD stepsize should be set as $\epsilon_n = (\tau_0 + n)^{-\beta}$, where n is the iteration number, so that it decays with iteration number. We will set $\tau_0 = 100$ and vary ϵ_n by varying β. Implement the regularized version of DSGD-MF using $\lambda = 0.1$, although λ should be provided as input to your program.

Note that you will not have access to the exact iteration number n to calculate ϵ_n while performing SGD updates within a stratum, since strata-specific SGDs are performed in parallel. You need to use an approximate version of the iteration number to calculate ϵ_n. In particular, you should use $n = n' + \sum_{i,b} m_{bi}$, where $\sum_{i,b} m_{bi}$ is the total number of SGD updates made

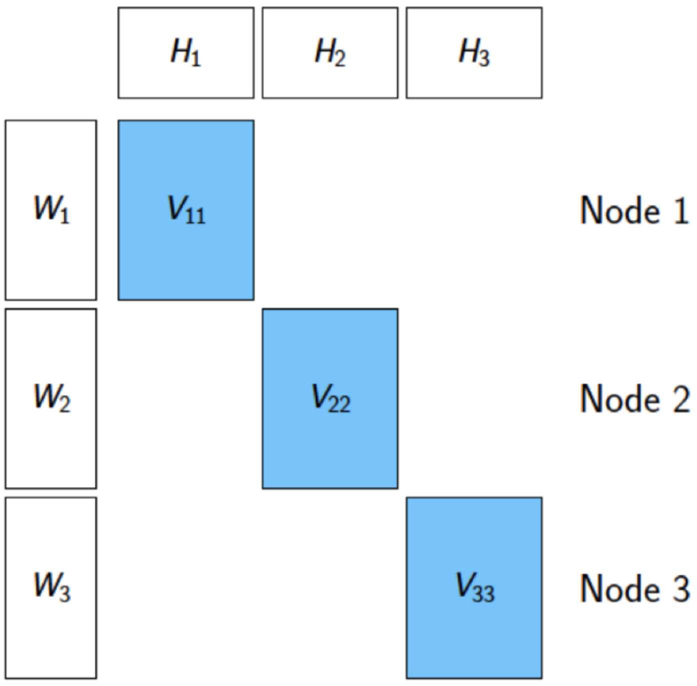

Figure 5.1: A stratum in **V**.

across all strata in all previous iterations, and n' is the number of SGD updates made by the current worker on its stratum so far. Thus, ϵ_n is synchronized for all workers at the end of every iteration, but is allowed to be calculated in a decoupled fashion once a worker starts performing SGD updates for the current iteration on its local stratum.

Also, remember to randomly initialize your factor matrices **W** and **H**; do not initialize them to zero matrices!

6 Deliverables

Create a folder in your repository named `assignment4`. Keep all your code for this assignment there; I will be using autograding scripts to check your progress, so make sure your repository is named correctly and precisely!

I will reference the code in that folder for partial credit. This needs to have a commit timestamp *before* the deadline to be considered on time. You should implement the algorithm by yourself instead of using any existing machine learning toolkit.

Please include a `README` file. In the `README`, document precisely how to run your program (I should be able to follow the instructions; this is how I will grade your assignment!) and any known bugs or problems you were unable to fix prior to the final submission.

You may use either Azure (if you have credits) or a UGA machine with at least 4 cores (8 threads). Please answer the following questions (reconstruction error L_2 refers to Eq. 1.3).

1. Set the number of workers $B = 10$, the number of factors $F = 20$, and the decay rate $\beta = 0.6$. Plot the reconstruction error L_2 versus the iteration number $i = 1, 2, ..., 100$. Include the plot, and explain any trends you observe.

2. Set the number of iterations $I = 30$, the number of factors $F = 20$, and the decay rate $\beta = 0.6$. Plot the runtime of your Spark code versus the number of workers $B = 2, 3, ..., 10$ in steps of 1. Include the plot, and explain any trends you observe.

3. Set the number of iterations $I = 30$, the number of workers $B = 10$, and the decay rate $\beta = 0.6$. Plot the reconstruction error L_2 versus the number of factors $F = 10, 20, ..., 100$ in steps of 10. Include the plot, and explain any trends you observe.

4. Set the number of workers $B = 10$, the number of factors $F = 20$, and the number of iterations $I = 30$. Plot the reconstruction error L_2 versus the decay rate $\beta = 0.5, 0.6, ..., 0.9$ in steps of 0.1. Include the plot, and explain any trends you observe.

5. Is there any advantage to using DSGD-MF instead of singular value decomposition (SVD), which can also be used to find a matrix decomposition for recommendation?

6. Explain clearly and concisely your method (as in, the method you used in the code you wrote) for creating strata at the beginning of every iteration of the DSGD-MF algorithm.

7. If you were to implement two versions of DSGD-MF using MapReduce and Spark, , do you think you will find a relative speedup factor between the two iterations (all other parameters assumed to be fixed)? Which implementation do you think will be faster? Explain why. If your answer depends on any general optimization tricks related to MapReduce or Spark that you know, please state them as well.

7 Bonus 1

Use a built-in (**not** distributed!) SVD solver for Java (Apache Commons), Scala (Breeze), or Python (SciPy) and compute the SVD directly with $k = 20$ principal components.

Now, run your DSGD-MF program, using number of workers $B = 10$, the number of factors $F = 20$, the decay rate $\beta = 0.6$, and the number of iterations $I = 30$. What are the reconstruction errors of the two methods? Is reconstruction error a fair comparison for these two methods?

8 Bonus 2

Use a distributed SVD solver for either Java, Scala, or Python to compute the SVD of the Netflix data for $k = 20$ principal components. Compute the reconstruction error.

Now, run your DSGD-MF program, using number of workers $B = 10$, the number of factors $F = 20$, the decay rate $\beta = 0.6$, and the number of iterations $I = 30$. What are the reconstruction errors of the two methods? Is there any way you could match the error rates of the two solvers? How?

9 Marking breakdown

- Code correctness (commit messages, program output, and the code itself) **[30 points]**
- Questions 1-7 **[10 points each]**
- Bonus 1 **[25 points]**
- Bonus 2 **[25 points]**

10 Other stuff

START EARLY. Yet again, there is not a lot of code to be written, but parsing all the math and truly grasping the concepts at work can take some time.

Microsoft Azure builds its HDInsight platform against specific builds of Apache Spark, and unfortunately the latest version available is 1.2.0 (Spark 1.5.1 was just released a couple weeks ago). There is an unofficial spark_azure script package for setting up custom Spark instances on Azure. I have not tested it, but you are welcome to give it a try.

Spark's performance, while generally much better than Hadoop, can vary wildly depending on how effectively you are caching your intermediate results (and what your memory strategy is). In particular, I highly recommend this chapter from *Learning Spark*, as it provides invaluable advice for partitioning your data in the most effective way possible so as to minimize network shuffling.

Spark has fantastic documentation for all its APIs.