MSCBIO/CMPBIO 2065: SCALABLE MACHINE LEARNING FOR BIG DATA BIOLOGY,
DEPARTMENT OF COMPUTATIONAL AND SYSTEMS BIOLOGY, UNIVERSITY OF PITTSBURGH

A2: Predicting Drug-Target Interactions with Large-Scale Matrix Factorization

DUE: Friday, November 11 by 11:59:59pm

Out October 21, 2016

In this assignment, your goal will be to predict drug-target interactions from a large-scale matrix factorization perspective. Drug-target interaction (DTI) is the basis of drug discovery and design, where the goal is to design exquisitely selective ligands against a single target. The drug discovery process is unfortunately time consuming and costly to determine experimentally. Additionally, the 'one gene, one drug, one disease' paradigm has been challenged in many cases. In its place the concept of 'polypharmacology' has been proposed for those drugs acting on multiple targets rather than one target. For example, serotonin and serotonergic are drugs that not only bind to G protein-coupled receptors (GPCRs), but also might bind to an ion channel. Such polypharmacological features of drugs enable us to understand the side effects of drugs or find new uses for drugs in the form of 'drug repositioning'.

Several *in silico* methods have been developed to address the issues of drug-target interaction (DTI) prediction and drug repositioning. The conventional methods are either ligand-based or receptor-based. However, these methods are typically target-focused and do not use information from known interactions with other targets. More recently, several network-based and phenotype-based methods were developed for such purposes. For example, a bipartite graph composed of US Food and Drug Administration (FDA)-approved drugs and proteins linked by DT binary associations can be constructed to gain a network centric view of DTIs. Recently, two successful examples of drug repositioning based on public gene expression data have been reported. In this assignment, you will consider DTI as a problem of large-scale matrix factorization. You will work with three different datasets, where you will predict one of these variables: EC50 values, growth inhibition values, or mRNA expression data. The first two datasets consider the effect of drugs on cell lines. The third does not involve drug interactions, but provides a large and dense dataset of biologically meaningful data.

1 DATA

You will test your code on three data sets of various sizes. For each, the input data file is a CSV file where each line consists of `(row,column,value)` as shown below.

```
145,17,0.288144
108,10,0.435161
430,11,1.534598
397, 5,7.524202
320, 9,0.111463
448, 2,0.552440
....
```

The lines are in no particular order.

Your program will also take a second file where each line consists of `(row,column)` indices. These are the missing values you must predict. You will write out (to a file, not the screen) the predicted values using the same format as the input data `(row,column,value)`. Since each line includes the row and column indices, you may write these in any order. The input data is stored on the Hadoop file system in 1MB chunks at a 12X replication factor.

Input files can be found here. The datasets come in three flavors:

Small Dataset: This is pharmacological profiling data from the Cancer Cell Line Encyclopedia. 504 different cell lines (rows) were tested against 24 cancer drugs (columns). The value is the EC50. The dataset has 7,128 values (41% are missing).

Medium Dataset: This is growth inhibition data from the National Cancer Institute. 52578 chemical compounds (rows) were tested against 159 cell lines (columns). The value is the GI50 (concentration at which 50% growth inhibition is observed). The dataset has 1,760,995 values (79% are missing - including 14% which we threw out because they exhibited no growth inhibition).

Large Dataset: This is mRNA expression data from the Cancer Cell Line Encyclopedia. The expression levels of 18,988 genes are characterized across 1036 cell lines. The dataset has 19,474,853 values (1% are missing - these are values we are withholding for testing; this is a dense dataset).

2 IMPLEMENTATION

Our intention is for you to implement some form of stochastic gradient descent (SGD) for filling in the missing values. You may choose the distributed stochastic gradient algorithm discussed in the class (and explained in more detail in the next section) or the literature. As a simple starting point, consider this (non-stochastic and non-distributed) implementation of matrix factorization which clearly shows how to translate the math into code.

As a starting point, consider iteratively applying Spark's SVD routine. In each iteration, the matrix reconstructed from the leading k singular vectors and values from the SVD decomposition is used to fill in the missing values of the data matrix This is not an efficient approach (particularly for sparse matrices), but it let's you leverage the Spark MLLib routines and so you may find it easier to code. You may use any of the built-in Spark routines. However, be aware that the built-in gradient optimization is not useful for this assignment.

You are encouraged to parameterize your code to maximize performance/accuracy on each dataset. Note that memory should not be (to much of) an issue with this assignment. The challenge is to properly parallelize your code, implement an efficient gradient descent algorithm with good convergence, and effectively parameterize your algorithm for the provided datasets (e.g., choosing a good k number of latent factors).

3 Stochastic Gradient Descent (SGD) for Matrix Factorization

If you are interested in implementing the distributed stochastic gradient descent for matrix factorization (DSGD-MF) discussed in the class, here is a link to the paper: Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent.

The DSGD-MF solution involves dividing the matrix into strata for each iteration and performing sequential stochastic gradient descent within each stratum in parallel. DSGD-MF is a fully distributed algorithm, i.e. both the data matrix $\mathbf{V}$ and factor matrices $\mathbf{W}$ and $\mathbf{H}$ 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[1] .

DSGD-MF considers minimizing sum of two local losses: plain non-zero square loss and the non-zero square loss with l_2 regularization:

$$L_{ij} = l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = (\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})^2 \tag{3.1}$$

$$L_{\text{NZSL}} = \sum_{(i,j)\in\mathbf{Z}} L_{ij} \tag{3.2}$$

$$L_2 = L_{\text{NZSL}} + \lambda\{\|\mathbf{W}\|_F^2 + \|\mathbf{H}\|_F^2\} \tag{3.3}$$

Algorithm 1 provides an overview of the first part. In it, we select a single datapoint and update the corresponding row of $\mathbf{W}$ and column of $\mathbf{H}$ in the direction of negative gradient. The gradients for L_{NZSL} (Eq. 3.2) loss are given as follows:

[1] Thanks to Dr. Shannon Quinn.

$$\frac{\partial}{\partial \mathbf{W}_{i*}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})\mathbf{H}_{*j} \tag{3.4}$$

$$\frac{\partial}{\partial \mathbf{H}_{*j}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})\mathbf{W}_{i*}^T \tag{3.5}$$

The gradients for L_2 loss (Eq. 3.3) are given as follows:

$$\frac{\partial}{\partial \mathbf{W}_{i*}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})\mathbf{H}_{*j} + 2\frac{\lambda}{N_{i*}}(\mathbf{W}_{i*})^T \tag{3.6}$$

$$\frac{\partial}{\partial \mathbf{H}_{*j}} l(\mathbf{V}_{ij}, \mathbf{W}_{i*}, \mathbf{H}_{*j}) = -2(\mathbf{V}_{ij} - \mathbf{W}_{i*}\mathbf{H}_{*j})\mathbf{W}_{i*}^T + 2\frac{\lambda}{N_{*j}}\mathbf{H}_{*j} \tag{3.7}$$

They're identical to the gradient equations 3.4 and 3.5 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 $\mathbf{W}$ and $\mathbf{H}$ is dependent on the entry $\mathbf{V}_{ij}$, and therefore the i^{th} row of $\mathbf{W}$ ($\mathbf{W}_{i*}$) and the j^{th} column of $\mathbf{H}$ ($\mathbf{H}_{*j}$). 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 $\mathbf{W}$ and $\mathbf{H}$ respectively from being updated by other workers.

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

4 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. 4.1 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 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 $\mathbf{V}$ and the factor matrices $\mathbf{W}$ and $\mathbf{H}$.

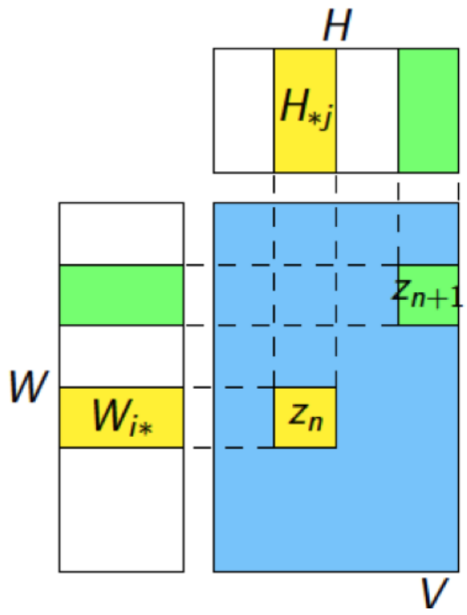

Figure 4.1: Element interchangeability in **V**.

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).

5 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). Algorithm 2 provides the pseudocode. As long as the parameter estimates $\mathbf{W}$ and $\mathbf{H}$ have not converged, we choose a set of strata, which creates disjoint blocks of $\mathbf{V}$, $\mathbf{W}$ and $\mathbf{H}$ 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 $\mathbf{V}_{I_n J_n}$ handed to that worker during iteration i. The worker will sequentially go through all the non-zero entries in its block $\mathbf{V}_{I_n J_n}$ and perform an SGD update for each entry as explained in SGD-MF earlier.

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

6 Parameters for DSGD-MF

You have to 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. You can try setting $\tau_0 = 100$ and vary ϵ_n by varying β. $\lambda = 0.1$ is a reasonable choice for the regularized version of DSGD-MF, although you may wish to experiment with other parameter settings.

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 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!**

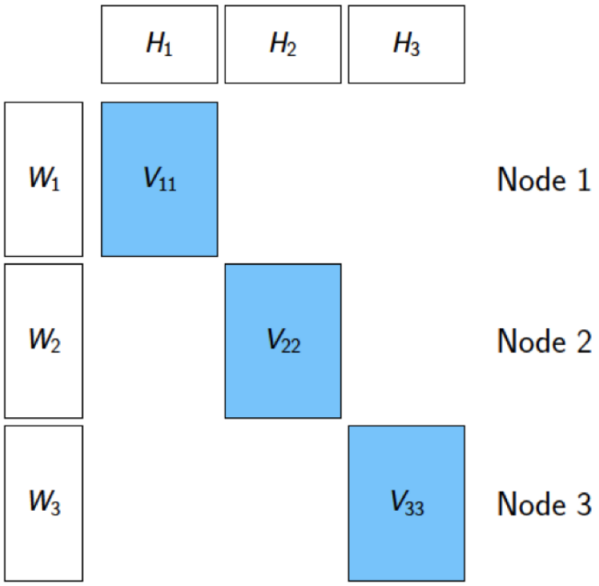

Figure 5.1: A stratum in **V**.

How do you know if you have a good DSGD-MF solution? First, in your iterative SVD implementation you will play with k, the number of number of principal (or singular) components, which is an estimate of the rank of the data matrix (and directly related to factor F in DSGD-MF). For each subspace size k you will plot reconstruction error L_2 versus the iteration number $i = 1, 2, \ldots$. This will help you understand the potential quality of the solution space for each of the datasets.

Second, you will play with the parameters of DSGD-MF, in particular, the number of workers B, the number of factors F, and the decay rate β and plot the reconstruction error L_2 versus the iteration number $i = 1, 2, \ldots$ for each dataset.

7 Submission

You should submit your code as a self contained Scala program Assign2 that takes three arguments: the input data file, the missing data file, and the name of an output file that you will write the predicted missing values to. Use a2_template.scala as a starting point for your code. Your code will be compiled:

```
scalac -classpath "$SPARK_HOME/jars/*" a2.scala  -d a2.jar -optimise
```

and submitted to spark:

```
$SPARK_HOME/bin/spark-submit --class Assign2 --conf
```

```
spark.serializer=org.apache.spark.serializer.KryoSerializer
a2.jar small.csv small_missing.csv small.out
```

We will run your code on a cluster of 26 quad-core machines. Each machine has 4GB of memory, 2.7GB of which is available to Spark. **Your program must finish processing a file within 30 minutes or it will be killed.**

8 EVALUATION

We will evaluate your program both in terms of how fast it is and how accurately it predicts missing values. For each of the datasets we have kept back a random selection of ~1% of the data. The score of your predictions is the average squared error of your predictions with respect to the true values. A lower error is better.

9 OTHER STUFF

START TODAY. Spark has documentation for all its APIs. As you know by now, Spark's performance can vary wildly depending on how effectively you are caching your intermediate results (and what your memory strategy is). You may also find this chapter from *Learning Spark* useful, as it provides advice for partitioning your data in the most effective way possible so as to minimize network shuffling.