Parallelized Probabilistic Matrix Factorization

OU Kaishen
Department of Computer Science & Engineering
CUHK
1155089150@link.cuhk.edu.hk

DENG Weijun
Department of Computer Science & Engineering
CUHK
1155084561@link.cuhk.edu.hk

PENG Yibo
Department of Computer Science & Engineering
CUHK
1155084518@link.cuhk.edu.hk

GUAN Chenye
Department of Computer Science & Engineering
CUHK
1155089134@link.cuhk.edu.hk

ABSTRACT

In this paper, a novel and efficient algorithm [1] for large scale matrix factorization is studied and implemented. Parallelized Probabilistic Matrix Factorization (PPMF) is successfully designed and programmed on top of the Hadoop [2] MapReduce framework. The PPMF is realized by a strategy of division and Stochastic Gradient Descent (SGD). Using the regular MovieLens' dataset [3] as performance evaluation, we get a pleasant Root Mean Square Error (RMSE) of 0.77. The implementation details are also discussed in this paper.

Keywords

PPMF, SGD, MapReduce, Hadoop, RMSE.

1. INTRODUCTION

Nowadays, recommendation system is one of the most representative applications of the data analysis. Among various techniques in recommendation system, matrix factorization is considered as the most powerful and accurate method. In particular, Probabilistic Matrix Factorization (PMF), which are capable of factorizing partially observed matrix, attracts a lot of attentions. As a method following the maximum a posteriori (MAP) criterion, PMF owns a similar objective function with Singular Value Decomposition (SVD) and has a firm theoretical basis. However, facing big data, PMF becomes very embarrassed because large scale data matrix cannot be read in memory. Worse still, it has been proved that computing PMF in parallel is a great challenge. This challenge is exactly what we want to solve in this paper. The innovation we put forward is to implement PPMF in MapReduce framework. Hadoop, as an open source implement of MapReduce framework, is utilized in this project. This project aims to implement PPMF in an abstract sense, rather than for a certain application problem. In this way, our PPMF algorithm can be applied to extremely broad scope of applications.

2. DATA PREPROCESSING

Before we implement our PPMF algorithm in Hadoop, we need to preprocess our dataset first. The dataset we use is "ml-100k" and "ml-20m" form MovieLens. Ten percent of the dataset is extracted to calculate the RMSE.

To begin with, inconsistent indices of movies need to be handled. Because there will be many columns that are all zeros in the matrix if we use the original movie IDs. Apparently, we must renumber our movies. We sort the old indices and then map the old indices to a set of new and consistent indices. For example, the index sequence 1,3,5,7,9…12001 will be transformed to 1,2,3,…N, where N is the length of the sequence.

In PPMF, an original matrix Z will be factorized into two much smaller matrix W and H.

$$Z_{M\times N} = W_{M\times K} \times H_{K\times N}$$

The dimension K, which is much smaller than N and M, is also known as latent factor. Latent factor tells the hidden information inside the dataset. Random float number, which is varying from 0 to 1, will be employed as the initial value of W & H. However, such a simple method may lead to a too large dot product between W_{i*} and H_{*j} when K is large. So we divide each float number by the square root of K, which means the dot product between each two vector will be restricted to $\frac{1}{K}$.

Last and the most important part is splitting dataset and parameter matrices. This is the preparation for parallelized computation and parameter updates.

For dataset Z, we need to split it into $d \times d$ blocks, each block Z_{IJ} contains $\frac{M}{d}$ rows and $\frac{N}{d}$ columns, except for those containing the last row or the last column. This procedure is shown in Figure 1. Given a pair (userId, movieId), we use the new pair $([\frac{userId}{ceil(\frac{M}{d})}], [\frac{movieId}{ceil(\frac{N}{d})}])$ to show which block it belongs to. The ceil function ceil(x) is the smallest integer larger than or equal to x. The floor function [x] is the largest integer less than or equal to x. For the parameter W and H, as shown in Figure 2, we also split them into d blocks.

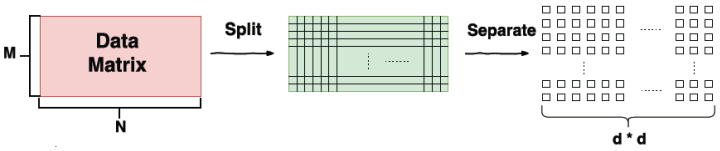

Figure 1. Matrix Splitting.

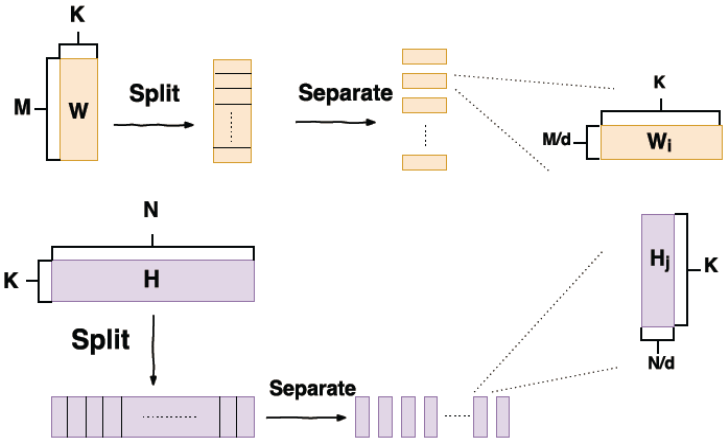

Figure 2. Parameters Splitting.

3. ALGORITHM

After splitting our dataset Z into $d \times d$ blocks and W & H into d blocks, matrix factorization is easy to be parallelized. We implement our algorithm in MapReduce structure. The total reduce procedure should deal with d blocks. In each reducer, we select one block Z_{IJ} from original $d \times d$ blocks. The block I of W and the block J of H will be trained by these data. With careful design, we ensure there are no reducer will train the same parameters in case that the previous result will be overlapped by the latter. For simplicity, the row index is fixed while we shuffle the column index. A permutation of d is generated and bound with a continuous sequence to be the selection. For example, $(1,P_1), (2,P_2)\ldots(d, P_d)$ is a possible selection. This selection determines which blocks we choose and makes sure there is no overlap among them.

Within each reduce procedure, we open the Hadoop File System (HDFS) and retrieve the relevant parameters to train. Traditionally, the loss function or objection function of PMF is:

$$L = \frac{1}{2}\sum_{i=1}^{M}\sum_{j=1}^{N}\left(R_{ij} - U_i^T V_j\right) + \frac{\lambda_U}{2}\sum_{i=1}^{N}\left\|U_i\right\|^2 + \frac{\lambda_V}{2}\sum_{i=1}^{M}\left\|V_j\right\|^2$$

In our PPMF, part of the data is utilized to approximate the true loss function. Here $L_{(d, Pd)}$ is the local loss computed by $Z_{(d, Pd)}$.

$$L_{true} \approx d \times (L_{(1, P_1)} + L_{(2, P_2)} + L_{(3, P_3)} + \cdots + L_{(d, P_d)})$$

Based on the derivative rules,

$$L_{true}' \approx d \times (L_{(1, P_1)}' + L_{(2, P_2)}' + L_{(3, P_3)}' + \cdots + L_{(d, P_d)}')$$

In SGD, one data point (i, j) inside the block is randomly chosen to approximate the local loss function derivatives.

$$\frac{\partial L_{(d, P_d)}}{\partial W_{i*}} \approx \frac{\partial L_{(i,j)\in Z_{(d, P_d)}}}{\partial W_{i*}} \qquad \frac{\partial L_{(d, P_d)}}{\partial H_{*j}} \approx \frac{\partial L_{(i,j)\in Z_{(d, P_d)}}}{\partial H_{*j}}$$

The update rule of SGD, also shown in Figure 3, is

$$W_{i*} \leftarrow W_{i*} - \eta \frac{\partial L_{true}}{\partial W_{i*}} \qquad H_{*j} \leftarrow H_{*j} - \eta \frac{\partial L_{true}}{\partial H_{*j}}$$

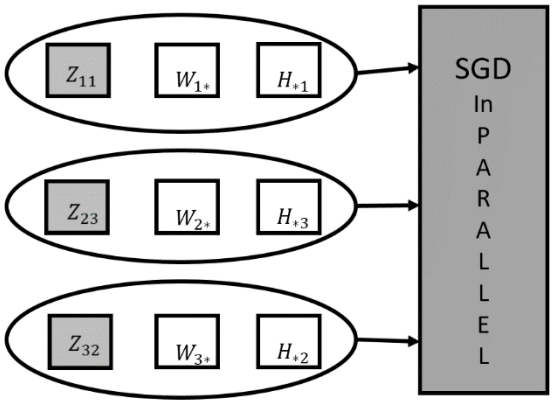

Figure 3.PPMF in cluster.

We need to pay attention to the learning rate η, which is the key parameter in the SGD algorithm. If η is too large, the result can hardly converge. On the other hand, if η is too small, the parameters will evolve too slowly to reach the optima. After numerous attempts, we find an appropriate method to update η while balancing the speed and convergence.

$$\eta \leftarrow \frac{\eta_0}{(\mathrm{iter} + 1) \times \alpha}$$

Where η_0 is a constant initial number, α is a constant factor and iter is the iteration number.

How to select data points in SGD is also very important. Two factors should be consider, randomness and coverage. Good result is obtained when (1) the selected training samples cover most part of the local block $Z_{(d,Pd)}$, and (2) the training sequence is randomized [1]. In our algorithm, due to limited time, we only focus on the randomness and select data point in completely random order. But there may exist duplicate data points, so the result is less accurate. The best approach is to generate a permutation of the local data set, which enables us to guarantee the coverage and randomness simultaneously.

4. IMPLEMENTATION DETAILS

The PPMF algorithm is implemented in MapReduce framework in Hadoop, which is one of the most popular platform for distributed computation.

4.1 MapReduce

MapReduce is a programming model for processing and generating large datasets, and it is amenable to versatile real-world tasks. [4]

In general, this computation model takes a set of input key/values pairs and produces multiple output key/value pairs. The user only need to implement two important functions, Map and Reduce.

Map function reads data in the form of key/values pairs and output zero or more intermediate key/value pairs. Reduce function accepts intermediate keys and corresponding value lists, performs computation, and eventually outputs final key/value pairs.

4.2 PPMF in MapReduce

The PPMF algorithm is implemented in original java Hadoop MapReduce library. Here, we use some tricks. The input and output files of Hadoop are just fakes for running the software. The real input and output files are manipulated inside the reduce function. To be more specific, we directly open and rewrite a file in HDFS during the reduce procedure.

As for dividing Z, the training data is loaded in Text input format. The Map function figures out which block the particular record belongs to, i.e. block id. Then it will set this block id as the intermediate key, and record as the intermediate value. For examples, the record (userId=1, movieId=1, rating=4.5) belongs to the block Z_{00}. The Map function would send "0_0" as the intermediate key and send the value of "1\t1\t4.5". Reduce function would accept key/values pairs and write the values into particular Z block files in HDFS. The Z block files' name are specified by the corresponding keys. For example, we write "1\t1\t4.5" into file named 0_0. In order to fulfill the MapReduce process, we also output one useless key/value pair.

As for W and H initialization, we give a fake input file to Hadoop. The content of this file is of no use so we read a very small file to increase the speed of the Map procedure. During the Reduce

procedure, W and H Matrix files with initial values will be created in HDFS, just like what we mentioned before.

For the training process, we still load the Z, W, and H Matrices in Reduce procedure. After SGD computation, updated W and H parameters are written back to HDFS. We train the model by doing MapReduce iteratively. In each iteration, the blocks' permutation is generated in Map function, and the actual training is done in Reduce function. Learning η will also be updated in every iteration, as mentioned before.

4.3 Parameter selecting

4.3.1 Tradeoff of reduce task number

A reducer is referred to the application of the Reduce function to a single key and its value list [5]. A reduce task receives one or more keys and their associated value lists. That is, a reduce task executes one or more reducers. By default, all reducers would be executed in one reduce task, which is not efficient when the computation power is enough. However, setting too many reduce tasks, especially when the number of reduce tasks is larger than the number of reducers, would not improve efficiency. This is because starting a reduce task itself has some overheads. In practice, we found that setting half of "d" as the number of reduce tasks would be a good choice.

4.3.2 Tradeoff of parameter d

Parameter "d" determines the number of blocks we divide and the percentage of total data we use for each iteration. If the "d" becomes larger, we have more reducers in training procedure, which benefits the efficiency. However, the data we use for each iteration become smaller, and more iterations are needed for final convergence. Worse still, this leads to more IO operations. In practice, we choose "d=5".

4.3.3 Tradeoff of dimension K

K is the latent factor which represents hidden information in the original matrix. In general, when the K becomes larger, W and H matrices would contain more hidden information and hence the factorization will be more accurate. However, when K is too large and there is not so much hidden information, the result would not be better at all. In practice, we set the "K" to 200 for the data set we use.

4.4 VMware for testing small dataset

It is too expensive to test the program on web clusters, like EC2 or EMR, so we configure a Hadoop cluster on three Linux Virtual Machines (one Master and two Slaves) on VMware. In this case, we first guarantee that our program could successfully run with small dataset. Then we run the large dataset on a real cluster with 20 workstations in our school.

4.5 Attempt on EC2 & EMR

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers [6]. Amazon EMR, referring to Amazon Elastic MapReduce, is an upgraded version of EC2. Although much time and effort have been devoted into setting the EMR, we fail to rum the program on Amazon. Because we build our project on HDFS, but what EMR provide is the S3 file system API. Therefore, we turn to use a cluster in our school.

The timeline of our project is shown in the Figure 4.

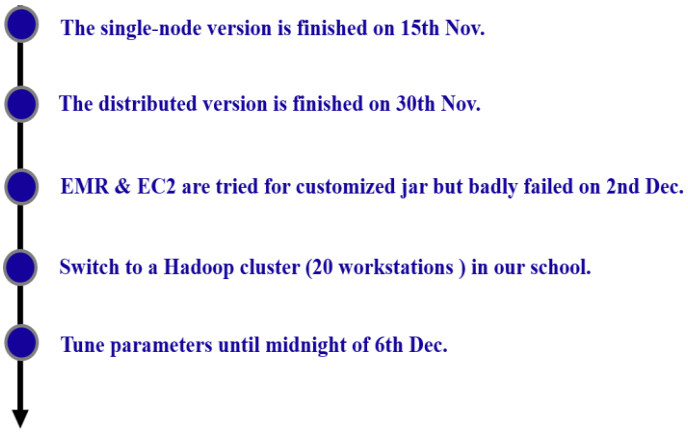

Figure 4. Timeline.

5. RESULT

A pleasant RMSE of 0.77 is obtained when the K is 200.

- ml-100k:

K	*RMSE*
30	0.99
50	0.96
100	0.94

Figure 5. RMSE for ml-100k.

- ml-20m:

K	*RMSE*
100	0.78
200	0.77

Figure 6. RMSE for ml-20m.

6. CONCLUSION

In conclusion, we did five things in this whole project.

1. Review and summarize the works that predecessors have done on PMF, Parallel Computing, and MapReduce on Hadoop;
2. Preprocess the dataset;
3. Program the traditional PMF with Java by ourselves and validate them by calculating RMSE;
4. Implement the PPMF on a multi-node Hadoop cluster and analyze the results.
5. Try and tune hyper-parameters to achieve a good RMSE.

7. ACKNOWLEDGMENTS

We would like to express our sincere gratitude to our instructor, Michael Lyu, for his teaching and kind support. By taking this course, Big Data Technology and Applications, we have learned many useful algorithms about big data. Also, this project gives us a good opportunity to practice handling a problem in large scale data. We improve ourselves a lot by cooperating with our teammates. These are very useful and beneficial to our future career and studies. Furthermore, we also want to express our sincere thanks to tutor, Yuxin Su, who gives us much guidance on this course and the project. Without his help, we cannot finish our work in a right way

8. REFERENCES

[1] Gemulla, Rainer, et al. "Large-scale matrix factorization with distributed stochastic gradient descent." Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2011.

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

[3] http://grouplens.org/datasets/movielens/.

[4] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113

[5] Leskovec, Jure, Anand Rajaraman, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2014.

[6] https://aws.amazon.com/ec2/?nc1=h_ls