Matrix Factorization with SGD using Spark and Scala

Linquan Chen, Yuankun Chang
Electrical and Computer Engineering, CMU

Abstract
Spark is an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster. This paper focuses on using Spark Programming Framework to optimize the sequential matrix factorization using Stochastic Gradient Descent(SGD) algorithm. Based on the sequential program, we implemented a basic programming using spark and analyzed the disadvantages of this version. Then, we optimize the program by updating the blocks in parallel, partition and group the blocks, and also tuning the spark configuration parameters to improve the performance. We also implemented the project using PySpark and Scala, and compare the performance of these two implementations.

Keywords: Matrix Factorization, SGD, Spark, Scala

1. Overview

Low-rank matrix factorization has recently been applied with great success on matrix completion problems for application likes recommendation system, link prediction for social networks, and click prediction for web search[1]. However, as this method is applied to larger and larger datasets, such as recommender systems like Amazon and YouTube, the data management quickly become challenging and facing a bottleneck. So we want to focus on this topic and try to improve the speed of matrix factorization.

Matrix factorization is a factorization of a matrix into a product of matrices. For example, we have a M-by-N data matrix Z, we can use two matrices W and H, whose size are M-by-K and K-by-Q, to estimate Z, such as W * H = Z. And K is a parameter that's typically referred to as rank.

We can use sequential Stochastic Gradient Descent(SGD) algorithm for matrix factorization. This is an iterative algorithm and we can refine the values of the two matrices by iterating. And for a right result, the factor matrices will not change a lot when iterate for several times. Therefore, we can utilize Apache Spark programming framework to accelerate the speed.

1.1 Matrix Factorization

Matrix factorization is a task that factorizes a M-by-N data matrix Z (with missing entries) into two factor matrices W and H of size M-by-K and K-by-N so that the product of M and N approximates Z, where K is a parameter that's typically referred to as rank. A larger rank allows the product of M and N to approximate Z more precisely but may cause overfitting. A sequential SGD algorithm for matrix factorization starts by randomly initializing W and H and then refining their values using an iterative algorithm. In each iteration, all known entries of Z are used to update the factor matrices and they are processed one by one. When processing each entry, the algorithm reads one row from W and one column from N (indexed by the coordinates of the data entry) and updates the read row and column.

This iterative algorithm ends when the factor matrices stops changing – when we say the algorithm has converged. We were at first given a sequential implementation of this algorithm, in which for one iteration, the data entries can be processed in any serial order. No matter how large the matrix is, no matter how many independent sub-blocks are lying idle waiting when other blocks are executing, a sequential implementation takes into no consideration of parallelization and thus take no advantage of pipe lining. it is thus greatly meaningful to have an efficient implementation of this algorithm. I will take advantage of a current distributed in-memory data processing framework, Spark.

1.2 Stochastic Gradient Descent

Stochastic gradient descent (often shortened in SGD) is a gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions. In matrix factorization, we will initially create two result matrixes H and W in a random way. Then we keep adjusting the values of each line in H and W to make the product of them approaching the corresponding value in the original matrix M as close as possible.
Suppose given a spilt level N, we will divide the original matrix into N*N blocks, and thus divide H and W into N blocks correspondingly. In this way, mathematically we can calculate block (0,0) of original matrix by block 0 of H and block 0 of W, calculate (1,2) by block 1 of W and 2 of H, etc.

For each block of M, we can retrieve every node of it, which is a tuple (user_id, movie_id, value). We denote user_id and movie_id as x, y. With x and y, we can find this value in matrix M should be the product of row x of W and column y of H. The different between the product and the actual value will be used to adjust the values of the row and the column in W and H.

The drawbacks of sequential implementation is that it doesn't take advantage of parallelism although data are chunked into separates. As I mentioned before, in order to update block i in W and block j in H, we need the use of block (i, j) of M. In other words, when the iteration goes to block (i, j) we are updating ith and jth block of W and H, in which case all other blocks except i and j in H and M are sleeping idle. Taking advantage of parallelism can greatly improve efficiency of this algorithm. While we still need to avoid race condition by letting the iteration go to, for example block (i, j) and (i, m) at the same time. I will illustrate my strategy later in next section.

This paper is organized as follows. Section 2 describes the Related work of matrix factorization and Spark's programming model. Section 3 shows the techniques we used in this project. In Section 4, it records optimization process, including: datasets, baseline results, basic spark implementation and optimize the Spark implementation and implemented a Scala version to improve the performance. And end with a discussion and future plan in Section 5.

2. Related Work

In this part, we did some survey on matrix factorization and related techniques. Firstly, we find that matrix factorization techniques are usually more effective in doing recommendations, such as personalized recommendations we received on Amazon. So it mainly deals with large scale datasets. Secondly, we find that there are mainly two approaches for matrix factorization, stochastic gradient descent(SGD) and alternation least squares(ALS). Compared with ALS approach, SGD combines implementation ease with a relatively fast running time, so we choose to use SGD approach for our project.

Researchers have explored the feasibility of using MapReduce to parallelize the sequential SGD by creating variants of SGD algorithm. Generally, the idea in these algorithms is to partition the input matrix into d1 * d2 blocks, which are distributed in MapReduce clusters. Both row and column factors are blocked conformingly and these algorithms are designed such that each block can be processed independently in the map phase taking only corresponding blocks of factors as input. While some algorithms directly update the factors in map phase while others aggregate the factor updates in a reduce phase[4]. Such parallelized SGD algorithms for matrix factorization are on par in terms of convergence time while running-time is much faster and stable for large data sets.

Success of Distributed SGD has inspired researchers to explore Spark as a framework to solve this problem since matrix factorization inherently allows the intermittent result be reused. Infact[1], Sparkler is an extension to the spark framework , which works on large-scale matrix factorization. Deriving inspiration from this, we decided to use spark to solve the problem of matrix factorization.

3. Techniques

3.1 Spark

As we know, MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters[2]. However, this model can't suit for all applications, such as applications that reuse a working set of data across multiple parallel operations. At these cases, Apache Spark programming framework is a good choice. It can support iterative process and maintains the scalability and fault tolerance likes MapReduce. Spark achieves two main architectures for parallel programming: Resilient Distributed Datasets(RDD) and Parallel Operations on these datasets (invoked by passing a function to apply on a dataset).

RDD represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. User can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. Therefore, it is an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

There are several parallel operations can be performed on RDDs:

- *reduce*: Reduces the elements of this RDD using the specified commutative and associative binary operator.
- *collect*: Sends all elements of the dataset to the driver program.
- *foreach*: Applies a function f to all elements of this RDD.
- *map*: Return a new RDD by applying a function to all elements of this RDD.

3.2 Algorithms

As shown in Figure 1, we updated the blocks in the diagonal order, which means that we updated the block 1(red), block 2(yellow), block 3(black) and block 4(green) at same time respectively. We can know from the figure 1, every color blocks correspond to different parts of H and W blocks, therefore we can update the H and W matrices parallel and not affect each other.

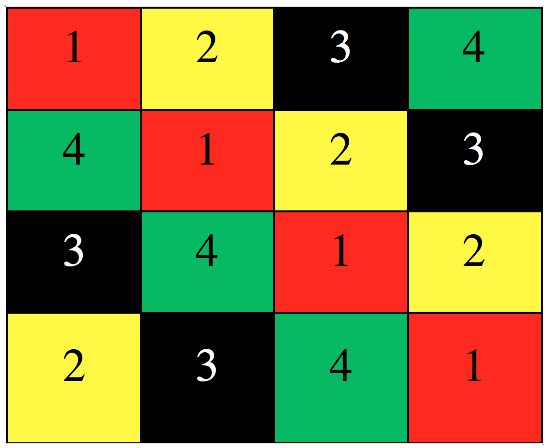


Figure 1. Updating blocks in diagonal

Considering the characteristic of RDD, we divided the Z, W and H into different blocks based on core num. Because the Z matrix will be used frequently and we divided it into N * N blocks based on maximum user id and maximum movie id. And for W and H matrices, they correspond user and movie, and we divided them into N blocks respectively.

As for number N, we decided it based on the total cores of slaves. In order to utilize the resource in nearly 100%, we treated one core as one process and update one block. Therefore, the core number is equally to partition number. For example, we built a cluster with one master and two slaves, every slave has four cores and the total cores is eight, so the partition number is eight.

3.3 PySpark and Scala

Spark currently supports multiple programming languages, such as Python, Scala and Java. It is really important to decide what language to choose for Spark project, because different language will result in different performance. Because our sequential code is implemented in Python, therefore we selected to use PySpark to implement the project first. After we dug deeply into the Apache Spark Framework, we found that Scala is a better choice for Spark project, not only considering the better performance when using Scala[5].

Firstly, Spark is implemented in Scala[6], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ[7]. Thus being proficient in Scala helps you digging into the source code when something does not work as you expect.

Secondly, Scala is based on JVM so it's native for Hadoop. Hadoop is important because Spark was made on the top of the Hadoop's filesystem HDFS. Python interacts with Hadoop services very badly, so developers have to use 3rd party libraries. Scala interacts with Hadoop via native Hadoop's API in Java. That's why it's very easy to write native Hadoop applications in Scala[8].

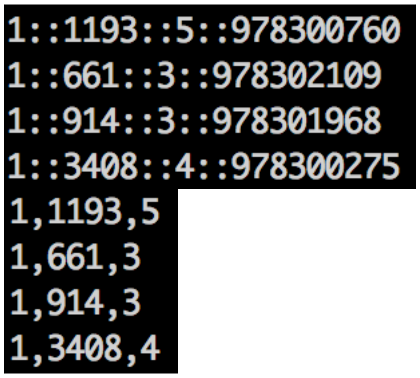
Based on above differences between Python and Scala, Scala is general faster than Python and it is also a better a choice when you need to learn more about Spark. Therefore, we also implemented a Scala version to deal with this problem.

3.4 Preprocess Data

We will use a few different sizes of the MovieLens datasets, each has millions of movies ratings made by hundreds of thousands of anonymous users of tens of thousands of movies. You can access the datasets at http://grouplens.org/datasets/movielens.

The format of the raw data is shown at Figure 2a. The format of the processed data is shown at Figure 2b. We used several different scripts to preprocess the raw data, such as following:

```
unzip ml-1m.zip ; cut -d':' -f1,3,5 ml-
1m/ratings.dat | sed 's/:/,/g' > rat-
ings_1M.csv
```



```
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
```

```
1,1193,5
1,661,3
1,914,3
1,3408,4
```

(a) **(b)**

Figure 2. Format of raw data and processed data

4. Result Analysis

4.1 Dataset

We got three different datasets(ML_1M, ML_10M, ML_22M) from MovieLens, and in order to improve the optimization speed, we created a smaller dataset ML_5M from dataset ML_10M. The details of every dataset is listed at Table1, which including ratings, users, movies, and used rank when doing matrix factorization.

Table1. Details of every dataset

Dataset	Ratings	Users	Movies	Rank
ML_1M	1000000	6000	4000	30
ML_5M	5000000	36000	5000	60
ML_10M	10000000	72000	10000	100
ML_22M	22000000	240000	33000	500

4.2 Sequential Implementation

We implemented the sequential python code for matrix factorization, which is used as baseline of the project. We run the sequential python program on four different datasets and got the result at Table2.

Table 2. Results of sequential python program

Dataset	Runtime (seconds)	Runtime memory[a]	rmse
ML_1M	360	2250	0.896
ML_5M	2000	15000	0.875
ML_10M	4800	36000	0.864
ML_22M	36000	270000	0.859

[a] runtime memory = provisioned memory * runtime, provisioned memory is 7.5G at this time

4.3 PySpark Program

Based on the sequential complete, functional(Python) implementation, we utilized the spark programming to implement the algorithms. we read the data file and used RDD to store the data, also using cache. As for H and W matrices, we used two RDDs to store the values. At last, the most important part-iterative, we just followed the implementation provided and updated one block at one time. Also, because updating a part of the RDD is very expensive, so we will transfer the RDD(W and H) into list at every slave, and updated this list every time. After updating, the driver will collect the updated W and H, and parallelize them into RDD again.

After doing experiment, we found this implement is not very efficient, shown at Table3. Because we were bounded by the sequential implementation. In fact, although we used RDD to store the reused data, the program framework is still sequential and wasted lots of resource in updating process, such as transfer the RDD into two dimensional array and collect the data between slaves and drivers, also rebuilt the RDD again based on the updated W and H.

Based on the above problems, we optimized the programing in three aspects: Firstly, we updated N(the partition number) blocks at same time instead of one block; Secondly, we used partitions on specific RDDs to distribute them across slaves, and grouped Z, W and H blocks; Thirdly, we tuning the spark configuration parameters to improve the performance.

Then we doing the same experiments on two datasets, the results are shown at Table4. After we compared the results of basic PySpark program and parallel PySpark program, we found this increases the speed up from 1.25 to 3.46 for ML_1M, and from 1.28 to 1.80 for ML_5M. This is Really a huge improvement.

Table 3. Results of basic PySpark program[b]

Dataset	Runtime (seconds)	Runtime memory[a]	rmse
ML_1M	286	12870	0.896
ML_5M	1560	70200	0.866

[a] runtime memory = provisioned memory * runtime, provisioned memory is 45G at this time
b Cluster details for the experiment: 1*m4.xlarge + 2*m4.xlarge

Table 4. Results of Parallel PySpark program[b]

Dataset	Runtime (seconds)	Runtime memory[a]	rmse
ML_1M	104	4680	0.893
ML_5M	1113	50085	0.865

[a] runtime memory = provisioned memory * runtime, provisioned memory is 45G at this time
b Cluster details for the experiment: 1*m4.xlarge + 2*m4.xlarge

4.4 Scala Program
At this time, we implemented the Scala program based on the same algorithms above and used the same datasets to do the test. We found that we can speed up the performance just using the Scala to program and without anything else.

Table 5. Results of Scala program[b]

Dataset	Runtime (seconds)	Runtime memory[a]	rmse
ML_1M	75	3375	0.892
ML_5M	854	38430	0.864

[a] runtime memory = provisioned memory * runtime, provisioned memory is 45G at this time
b Cluster details for the experiment: 1*m4.xlarge + 2*m4.xlarge

4.5 Optimization
There are some parameters that are significant to improve the performance of the program.

4.5.1 Partition Number
This is not a parameter for spark configuration but a parameter in the program. Interestingly, I found this parameter will also affect the performance of the program. Specifically, the higher the value is, the program runs faster. The following is my experiment results, shown at Table6.

Table 6. Compare Partition Number[b]

Partition number	Runtime (seconds)	Runtime memory[a]	rmse
8	854	38430	0.864
16	683	30735	0.865
32	605	27225	0.864
64	666	29970	0.864

[a] runtime memory = provisioned memory * runtime, provisioned memory is 45G at this time
b Cluster details for the experiment: 1*m4.xlarge + 2*m4.xlarge; Dateset: ML_5M.

We found that the program runs faster when increasing the partition number, but there is tradeoff of this parameter. Because when there are two many partitions, it will cost more time to collect the partitions. Therefore, for our architecture which has one master and two slaves, we choose 32 as our best partition number.

4.5.2 Cluster Type
Until now, we spent a lot of time to improve the runtime performance. However, we ignored the memory usage, and the fact is that our memory usage is pretty low. After we analysis our cluster, we found there is a huge waste of the memory resources. As we known, the memory of the m4.xlarge is 16G, but we waste the majority part of it. After we studied the AWS EC2 resources, we decided to use c4.xlarge instead, whose memory is only 7.5G and it is also computing optimization. The compare results are shown at Table7.

Table 7. Compare Instance Type[b]

Instance Type	Runtime (seconds)	Runtime memory[a]	rmse
m4.xlarge	605	27225	0.864
c4.xlarge	539	10241	0.865

[a] runtime memory = provisioned memory * runtime, provisioned memory for m4.xlareg is 45G at this time, c4.xlarge is 19G
[b] Partition number: 32, Dateset: ML_5M, one master and two slaves

We found that after changing the instance type of the cluster. Not only speeding up the runtime, but also increasing the memory usage, from 27225 to 10241, which is better than the origin one.

4.5.3 Spark Parameter
There are two really important parameters: --executor-memory and --num-executors. we noticed that maybe there is a bug for the new version of EMR. When we set the --num-executors equals to 2, but we always only found one executor working. we think there may have a bug for calculating the executors. Therefore, we need to lower the --executor-memory. For example, when the memory of slave is 15GB, we set the --executor-memory

to 10GB. And for --executor-cores, because the slave also need resource to run the operate system, so we need to leave one to two cores for the system.

Other important parameters are relative with driver, such as --driver-memory and --drive-cores. Because the drive node not only run the driver program, but also run the YARN and Application Manager, therefore, we can allocate all resources to driver. For example, if the driver node has 15GB memory and 4 cores, we will set the --driver-memory to 8GB and --driver-cores to 2.

The most challenge is to run the 22M data with rank 500, we had account lots of problems relative to memory. Firstly, the process is out of memory, so we change the instance to r4.xlarge whose memory is 30 GB. After that, we counted a problem that result size exceed the maximum return size of driver, therefore, we tuned the parameter --driver-maxResultSize to 2 GB(the default is 1 GB). Then we come across a problem that the memory exceeds the physical memory, So we tuned the –executor-memory, and utilize the memory mostly. However, it broke the heap after running for two hours.

When we use the Scala program, it can't run on the cluster contains c4.xlarge instances, because it will have the problem of out of memory, still can not fix this. But it can successfully run on the cluster contains one m4.xlarge master node and two m4.xlarge slave nodes.

4.5.4 Best Results

Based on the above work, we used the Scala problem with 32 partitions as our final best program. And we run the program on the cluster which consist of one c4.xlarge master node and two c4.xlarge slave nodes. The final results in shown at Table8. And we also calculated the speed up of runtime and memory usage, which is shown at Table9.

Table 8. Best results of Scala Program[b]

Dataset	Runtime (seconds)	Runtime memory[a]	rmse
ML_1M	70	1330	0.892
ML_5M	539	10241	0.865
ML_10M	1027	19513	0.863
ML_22M[c]	19894	895230	0.860

[a] runtime memory = provisioned memory * runtime, provisioned memory is 19G at this time
[b] Cluster details for the experiment: 1*c4.xlarge + 2*c4.xlarge; Partition Number: 32
c Cluster different: 1*m4.xlarge + 2*m4.xlarge, because c4.xlarge cluster will memory problem

Table 9. Speed Up of the Final Program

Dataset	Runtime Speed Up	Memory Usage	rmse
ML_1M	5.14	1.69	0.892
ML_5M	3.71	1.46	0.865
ML_10M	4.67	1.84	0.863
ML_22M	1.81	0.30	0.860

5. Conclusion

5.1 Problem

Based on the above analyze, the most challenge of my program is about memory. we think there are two steps to solve the problem: Firstly, we need to optimize my program. Because, we have created many large RDDs during the iterative processes. we need to discard them when no longer need them. Therefore, we need to re-analyze the process and optimize the memory utilization. Secondly, we can optimize the program at the aspect of tuning Spark Configuration parameters, such as cluster instance type and number, executor memory and number, driver memory and number, memory fraction and so on.

5.2 Future Plan

In the next step, we want to combine spark and redis. As we said in the abstract report, in theory, redis can accelerate Spark performance by up to 50 times. And we also know there is a Redis-Spark connector that expose the Redis data structures and API to Spark. Since we used python to write spark in the previous work, we are working on translating it into scala at this time. Hopefully, we could work it out.

Reference

[1] Li, Boduo, Sandeep Tata, and Yannis Sismanis. "Sparkler: supporting large-scale matrix factorization." Proceedings of the 16th International Conference on Extending Database Technology. ACM, 2013.

[2] Zaharia, Matei, et al. "Spark: Cluster Computing with Working Sets."HotCloud 10 (2010): 10-10.

[3]https://en.wikipedia.org/wiki/Collaborative_filtering

[4] Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent.

[5] https://datascienceschool.quora.com/Why-Scala-for-machine-learning

[6] Scala programming language.

http://www.scala-lang.org.

[7]Y.Yu,M.Isard,D.Fetterly,M.Budiu,Ú.Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In OSDI ’08, San Diego, CA, 2008.

[8] https://www.quora.com/Is-Scala-a-better-choice-than-Python-for-Apache-Spark