

Spark Implementation of Matrix Factorization with SGD

Xuerong Hu

xronghu1993@gmail.com

Abstract

This project will present an implementation of Matrix Factorization with SGD (stochastic gradient descent), an iterative optimization algorithm for improving the guess until the amount of improvement per iteration falls under a threshold. All computation will be executed in Spark cluster resides launched with AWS EMR. We will evaluate the computational result by both accuracy of the prediction and efficiency of the implementation, by using respectively RMSE value and time elapse.

1. Introduction

Matrix factorization is a task that factorizes a P-by-Q data matrix Z (with missing entries) into two factor matrices W and H of size P-by-K and K-by-Q so that the product of W and H approximates Z , where K is a parameter that's typically referred to as rank. A larger rank allows the product of W and H 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 H (indexed by the coordinates of the data entry) and updates the read row and column. [1]

This iterative algorithm ends when the factor matrices stops changing – when we say the algorithm has converged. We were at first given an 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.

In section 2, I will first introduce the basic idea of this algorithm and a brief walk through of how the sequential version of implementation works. Then in section 3, I will introduce my implementation in this project using Spark and how this idea differ from the sequential one to achieve an optimal efficiency. In the forth section, I will introduce the process of tuning the Spark Cluster

on Yarn resource management platform. In the last section, I will display my final run configurations and the execution result of each submitting.

2. SGD and Sequential Implementation

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 in writup is, 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 it and j th 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.

3. Spark Implementation

3.1 Fundamental ideas

In Spark programming, I will try to take use of parallelism as much as I can. I spark RDDs are computational units. All transformations on this RDD will be recorded until the next first action(Lazy Transforma-

tion). Every time the Spark computes on a single RDD, Level of parallelism depends on the number of partitions of this RDD. And normally I consider the number of parallelism as the total number of cores of slaves in this cluster.

The basic idea is very similar as what is was in the sequential implementation. Original matrix is decided into $N \times N$ blocks when H and W are both decided into N blocks. However, the implementation change a lot. Basic principles need to follow are:

1. Partition RDD that represents M into $N \times N$ parts, partition H and M into N parts using `partitionBy(number)` function
2. As I described at the end of section 2. parallel computation on M blocks that are in same line or column of M will cause updating conflict and thus reduce efficiency and accuracy. I thus traverse the blocks of M in diagonal way.
3. We have to eliminate the dependency between RDDs to improve parallelism and reduce network traffic during computation. I thus joined parts of M, H and W together before computing SGD on these blocks.
4. The joined resulting RDD J has to be partitioned by N for parallelism.
5. Cache RDD to improve efficiency and un-persist RDDs that does not need to be maintained in memory anymore.

3.2 Code detail

In this part of report, I will give a walk through of my code, and I will introduce in detail the critical parts of of the code that reflect the fundamental ideas I mentioned above.

Initializing

```
matrix_rdd=sc.textFile(brd_conf.input_path,
N).map(map_line).persist()

...

matrix_rdd = matrix_rdd.map(lambda x: ( (x[0]/x_block_dim.-
value)*N + x[1]/y_block_dim.value, x[0], x[1], x[2])).group-
By(lambda x:x[0],numPartitions=N*N).cache()
```

The original matrix M is first read into memory through `textFile()` API and then blockified according to the locations of each node. This part is just same as sequential implementation. However, one thing need to mention here is the block ID. Instead of using a value pair (x,y) as a block ID, I flatten it out. Like the matrix below.

0	1	2
0		

3	4	5
6	7	8

H

0	1	2
---	---	---

W

W and H are also initialized into N blocks with block id $0 \sim N-1$

```
H_rdd=sc.parallelize(range(N)).map(lambda x : (x,np.random.rand( y_block_dim.value, rank))).partitionBy(N)
```

```
W_rdd=sc.parallelize(range(N)).map(lambda x : (x,np.random.rand( x_block_dim.value, rank))).partitionBy(N)
```

SGD block by block

How to traverse through the blocks in M is key to this project. As we can see, to update block 0, in matrix H. We can use block 0,1,2 in matrix M. If we put block 0, 1,2 together into one RDD and do SGD, it will cause update conflict on matrix H block 0. Therefore in first iteration, I joined matrix M block 0, 4, 8 with H, W, and in second iteration I joined block 1, 5, 6 with H and M

....

In code, it looks like this

for delta in range(N):

```
matrix_part = matrix_rdd.filter(lambda line: ((line[0]%N -
line[0]/N == delta) or (line[0]/N - line[0]%N == N-
delta)) ).map(lambda line: (line[0]/N, line[1])).parti-
tionBy(N)
```

For matrix M, I first filter out the diagonal blocks and then change their block id in order to join with block H using block id.

Why do I need to change the block id? See example here: Assume it's the second iteration here, where I am now working on matrix M block 1,5,6. In which case I want block_M(1,5,6) to join with corresponding blocks in H and W, the join result should be looking like this [(M: 1, H: 0, W:1), (M:5, H:1, W:2), (M:6, H:2, W:0)]. I thus need to change block id of M and W (or H) because I am using block id as the key to do the join. In my implementation, I map matrix M's block id and W's block ids to the value equal to the H's blocks respectively.

After one iteration of updating H and W, I will re-generated the resulting the H and M RDDs for the use of next iteration.

```
H_updated=updated_HM_rdd.map(lambda x: x[0]).collect()
```

```
W_updated=updated_HM_rdd.map(lambda x: x[1]).collect()
H_dict.update(dict(H_updated))
W_dict.update(dict(W_updated))
W_rdd = sc.parallelize(W_dict.items()).partitionBy(N)
H_rdd = sc.parallelize(H_dict.items()).partitionBy(N)
```

As you can see, all computing centralized blocks are partitioned to improve efficiency.

4. Boosting up Spark Cluster

Two configuration files that I ever changed are
 /etc/spark/conf/spark-default-conf
 and
 /etc/hadoop/conf/yarn-site.xml

The first is used as default configuration of each spark application. Every time a new application is launched, these configurations are loaded into AM

The second is the default configurations of Yarn cluster. I tuned Yarn because I find sometimes EMR tend to allocate low memory to node managers who resides on slaves and is responsible for allocating containers inside of it.

One last part I tuned is the hadoop file system. Although it did not go as effective as I expected. But I think it still make some sense. I changed the block size of the input file. For example, for ratings_10M.csv file, which is 124 MB, I set the block size of this file to be 1.9375 MB because I have 8 cores in my cluster and I set partitionNum 8. The original block will be partitioned into 64 blocks. By changing the input block size, it will change the original partition number when the file is read into Spark.

4.1 Introduction to some key metrics & my settings & EMR's bug ?

spark.executor.memory: controls the executor heap size, but JVMs can also use some memory off heap, for example for interned Strings and direct byte buffers. The value of the spark.yarn.executor.memoryOverhead property is added to the executor memory to determine the full memory request to YARN for each executor. It defaults to $\max(384, .07 * \text{spark.executor.memory})$.

My setting: It's a very interesting part here ! Seems **EMR is having a severe bug** on calculating the max number of executors on a cluster. I used to set the executor memory to be a little less than the *yarn.nodemanager.resource.memory-mb*, and I had 2 instances, so theoretically I should have at least 2 executors. But I got only one. One student posted on piazza, found a reported bug of EMR describing the same issue. [[https://forums.aws.amazon-](https://forums.aws.amazon.com/thread.jspa?threadID=224258)

[com/thread.jspa?threadID=224258](https://forums.aws.amazon.com/thread.jspa?threadID=224258)]. Inspired by this, I changed the executor memory to be only half of *yarn.nodemanager.resource.memory-mb*, and I got 2 executors successfully running on my cluster.

yarn.nodemanager.resource.memory-mb: controls the maximum sum of memory used by the containers on each node.

My setting: I usually set it be to be 2.5GB less than the total amount of memory of that machine

yarn.nodemanager.resource.cpu-vcores: controls the maximum sum of cores used by the containers on each node. [2]

My setting: Usually set to be the full amount of cpu on that node

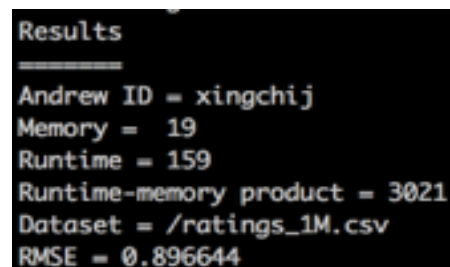
spark.broadcast.blockSize: Size of each piece of a block for broadcast. Too large a value decreases parallelism during broadcast (makes it slower); however, if it is too small, BlockManager might take a performance hit. Default value is 4M. [3]

My setting: When computing on ratings_1M.csv I set it to even smaller, 3M.

5. My Final Result and Settings

a. Ratings_1M.csv

```
3 x c4.xlarge
spark.executor.cores      4
spark.driver.memory       4620 M
spark.yarn.driver.memoryOverhead 512
spark.executor.memory     5120M
yarn.nodemanager.resource.cpu-vcores  4
yarn.nodemanager.resource.memory-mb 6120
```



```
Results
Andrew ID = xingchij
Memory = 19
Runtime = 159
Runtime-memory product = 3021
Dataset = /ratings_1M.csv
RMSE = 0.896644
```

b. Ratings_10M.csv

```
1 x c4.xlarge as master, 2 x m4.xlarge
spark.executor.cores      4
```

```
spark.driver.memory      5120M
spark.yarn.driver.memoryOverhead 512
spark.executor.memory     5120M
yarn.nodemanager.resource.cpu-vcores 4
yarn.nodemanager.resource.memory-mb 11246
```

```
Results
-----
Andrew ID = xingchij
Memory = 42
Runtime = 1661
Runtime-memory product = 69762
Dataset = /ratings_10M.csv
RMSE = 0.860311
```

Reference

- [1]. "Matrix Factorization with Spark"
<https://theproject.zone/student/writeup/15/107>
- [2]. "How-to: Tune Your Apache Spark Jobs (Part 2)" <http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/>
- [3]. "Spark Configuration" <http://spark.apache.org/docs/latest/configuration.html>