Fast MF for Online Recommendation with Implicit Feedback

Final Report

Doha Kaddaf, Honghao Yu, Jiayu Gan

1. Introduction

Recommender systems are crucial nowadays in delivering the best user experience for customers and increasing business sales. They are based on predicting the users rating of certain items based on past users' ratings of other items. The core purpose underlying all used techniques is to find hidden variables explaining users behavior called latent factors. Today we are exploring Fast Matrix Factorization (MF) for online recommendation with implicit feedback and implementing the Spark version of it.

The MF algorithms essentially boil down to factorising a user-item interaction matrix $R \in \mathbb{R}^{M \times N}$, where M and N denote the number of users and items respectively, into two matrices $P \in \mathbb{R}^{M \times K}$ and $Q \in \mathbb{R}^{N \times K}$ which denote the latent factor matrix for users and items respectively. Most MF algorithms use uniform weight for the missing data which is not the case in real world applications. Some authors have proposed to use weights based on the item's popularity. But such non-uniform weighting will result in an increase in computational complexity. For this purpose, an element-wise Alternating Least Squares (eALS) has been designed.

The eALS has evolved from ALS, which is a popular approach to optimize regression models such as MF and graph regularization. It works by iteratively optimizing one parameter, while leaving the others fixed. However, it has a significant bottleneck in that it requires matrix inversion operation. The eALS tackles this issue by optimizing each coordinate of the latent vector, while leaving the others fixed. It iteratively executes the update function for all model parameters until a joint optimum is reached[1].

By caching certain summation terms, the fast eALS not only allows a non-uniform weighting but also is K times faster than ALS, where K denotes the number of latent factors, which makes it more suitable for dealing with streaming online data. Plus it doesn't require using Stochastic Gradient Descent to learn the weights such as Devooght's method for example, bypassing the expensive determination of a good learning rate via a line search.

[1] He, X., Zhang, H., Kan, M. and Chua, T., 2016. [online] Comp.nus.edu.sg. Available at: <https://www.comp.nus.edu.sg/~xiangnan/papers/sigir16-eals-cm.pdf> [Accessed 31 March 2020].

2. Details of Map-Reduce Implementation

The fast eALS algorithm proposed in the article can be easily parallelized via Map-Reduce approach. As demonstrated on the right, the proposed algorithm is mainly made up of two nested loops. The first nested loop aims at updating parameters in matrix P (latent features for users) while the other aims at updating parameters in matrix Q (latent features for items). The inner loop (loop for each latent feature) must be implemented in sequential order since the update function (**Eq. 12** and **Eq. 13**) relies on a volatile variable which is generated in the previous iteration. However, the outer loop is parallelizable since the variables shared by iterations are either independent or constant.

Algorithm 1: Fast eALS Learning algorithm.

Input: $\mathbf{R}$, K, λ, $\mathbf{W}$ and item confidence vector $\mathbf{c}$;
Output: Latent feature matrix $\mathbf{P}$ and $\mathbf{Q}$;

Randomly initialize $\mathbf{P}$ and $\mathbf{Q}$;
for $(u, i) \in \mathcal{R}$ **do** $\hat{r}_{ui} \leftarrow$ Eq. (1) ; ▷ $O(|\mathcal{R}|K)$
while *Stopping criteria is not met* **do**
  // Update user factors
  $\mathbf{S}^q = \sum_{i=1}^{N} c_i \mathbf{q}_i \mathbf{q}_i^T$; ▷ $O(MK^2)$
  for $u \leftarrow 1$ **to** M **do** ▷ $O(MK^2 + |\mathcal{R}|K)$
    for $f \leftarrow 1$ **to** K **do**
      for $i \in \mathcal{R}_u$ **do** $\hat{r}_{ui}^f \leftarrow \hat{r}_{ui} - p_{uf} q_{if}$;
      $p_{uf} \leftarrow$ Eq. (12) ; ▷ $O(K + |\mathcal{R}_u|)$
      for $i \in \mathcal{R}_u$ **do** $\hat{r}_{ui} \leftarrow \hat{r}_{ui}^f + p_{uf} q_{if}$;
    end
  end
  // Update item factors
  $\mathbf{S}^p \leftarrow \mathbf{P}^T \mathbf{P}$; ▷ $O(NK^2)$
  for $i \leftarrow 1$ **to** N **do** ▷ $O(NK^2 + |\mathcal{R}|K)$
    for $f \leftarrow 1$ **to** K **do**
      for $u \in \mathcal{R}_i$ **do** $\hat{r}_{ui}^f \leftarrow \hat{r}_{ui} - p_{uf} q_{if}$;
      $q_{if} \leftarrow$ Eq. (13) ; ▷ $O(K + |\mathcal{R}_i|)$
      for $u \in \mathcal{R}_i$ **do** $\hat{r}_{ui} \leftarrow \hat{r}_{ui}^f + p_{uf} q_{if}$;
    end
  end
end
return P *and* Q

$$p_{uf} = \frac{\sum_{i \in \mathcal{R}_u} [w_{ui} r_{ui} - (w_{ui} - c_i) \hat{r}_{ui}^f] q_{if} - \sum_{k \neq f} p_{uk} s_{kf}^q}{\sum_{i \in \mathcal{R}_u} (w_{ui} - c_i) q_{if}^2 + s_{ff}^q + \lambda}. \tag{12}$$

$$q_{if} = \frac{\sum_{u \in \mathcal{R}_i} [w_{ui} r_{ui} - (w_{ui} - c_i) \hat{r}_{ui}^f] p_{uf} - c_i \sum_{k \neq f} q_{ik} s_{kf}^p}{\sum_{u \in \mathcal{R}_i} (w_{ui} - c_i) p_{uf}^2 + c_i s_{ff}^p + \lambda}, \tag{13}$$

2.1. Pipeline overview

The pipeline of our implementation of the Fast eALS algorithm is demonstrated below in **Figure 1**.

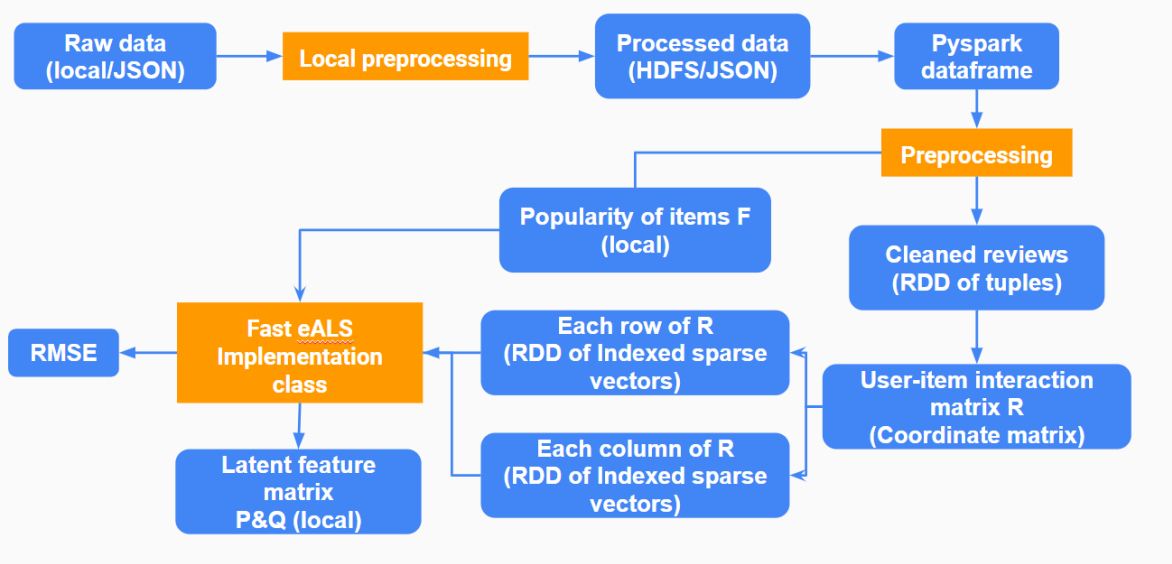


Figure 1. Pipeline of mapreduce implementation of Fast eALS algorithm

We experimented with the Yelp dataset which contains around 6.4M reviews. Please refer to **Table 1.** below for the descriptive statistics of the dataset. The raw dataset is encoded in JSON format and occupies 5.5 GiB space on local HDD. We had discarded irrelevant fields like “review text” and “review ID” and reorganized the JSON file locally on the laptop using a self-written python script that can read and write huge JSON files in stream. By doing so we managed to compress the file from 5.5 GiB to 1.0 GiB. The JSON file was then uploaded onto the cluster on cloud, stored in HDFS and imported into the pySpark environment with the help of the pyspark.read.json module. A native pySpark dataframe was built after importing JSON dataset, enabling pandas-style manipulation and SQL requiry simultaneously.

Table 1. Statistics of the Yelp dataset

Dataset	Review#	Item#	User#	Sparsity
Yelp	6,396,215	192,606	1,627,156	99.998%

For each user and business couple only the most recent data is selected using SQL in order to obtain the uniqueness of couples. Afterwards a numerical id is created for each user and each business. The purpose of doing so is to allow the usage of a coordinate matrix to represent the interaction matrix $\mathbb{R}$. A coordinate matrix requires having an RDD of tuples in this case each tuple is composed of 3 values : an integer corresponding to the business ID, another corresponding to the user ID and the rating itself.

The generation of an unique ID is done by listing all unique users IDs and business IDs. This can be done using SQL or RDD operations. The latter was chosen. At first two dictionaries were created to match every user's ID and business ID with their respective integer matches. A user defined function (udf) called “translate” was created to transform the columns into

their numerical counterparts. The udf “translate” takes as input the above mentioned dictionary and outputs the value corresponding to the key for each line.

The popularity of business f_i is computed using SQL then map reduce operations. At first a SQL query was created for counting the number of reviews per business. Using a mapping function the number of reviews power α is computed for each business ID, where α is an exponent controlling the significance level of popular items over unpopular ones. A Map and reduceByKey are used afterwards to sum the number of reviews power α for all businesses combined. This sum is needed in order to normalize the computed quantity for each business and to obtain finally a frequency. For this purpose a last mapping function is used to divide the quantity associated to each business Id by the sum.

The popularity of each item f_i and the interaction matrix $\mathbb{R}$ is all we need to derive from the dataset to feed the algorithm, whereas other quantities are either constant or temporary derivatives within the function scope. Thanks to the elegant implementation of coordinate matrix in pySpark, we can easily transpose the coordinate matrix $\mathbb{R}$ and reslice it into indexed sparse vectors along either row axis or column axis.
In general, the Fast eALS algorithm consists of two phases. In the first phase, the algorithm takes each row of the interaction matrix **R** (**R**[u,:]) as input and updates the corresponding row of the latent factor matrix for users **P** (**P**[u,:]). In the second phase, it takes as input each column of the interaction matrix **R** (**R**[:,i]), or each row of transposed matrix **R** (**R**.T[i,:]), and updates the corresponding row of the latent factor matrix for items **Q** (**Q**[i,:]) accordingly. On the other hand, the item popularity vector **f** was fed to the algorithm to generate the vector **c**, which denotes the confidence over a missing rating being a negative one.

In our implementation, the whole algorithm is broken down into first M then N parallel mini-tasks, as demonstrated in **Figure 2**. Note that the two phases cannot be executed in parallel since the **Alternative Least Square** algorithm updates the two latent factor matrices alternatively, in sequential order.

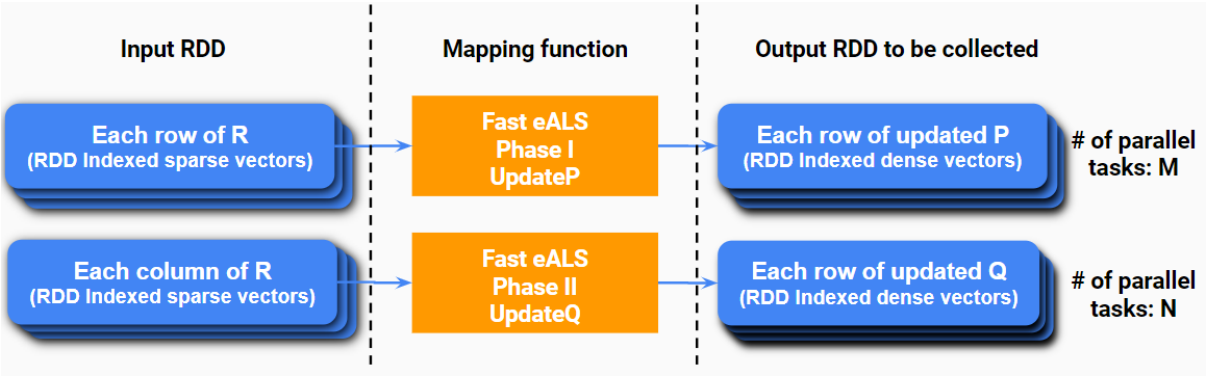


Figure 2. Parallelization of Fast eALS algorithm and RDD objects

To execute the first phase, an RDD object of indexed sparse vectors was created from the coordinate matrix R firstly, with each tuple representing each row of the interaction matrix R. The tuples are in the form of:

$$input\ tuple\ A : (u,\ (Sparse\ representation\ of\ u-th\ row\ of\ R))$$

Note that the sparse vector of R_u is the only variable transmitted to the mapping function via RDD tuple, while other essential variables are transmitted via either broadcasting or direct function call. A function UpdateP() is then applied to each of the tuples, converting them into the output RDD with the tuples in the form of:

$$output\ tuple\ A : (u,\ (Dense\ vector\ of\ u-th\ row\ of\ updated\ P))$$

The output RDD is collected on the driver node and the local matrix P is then updated row by row based on the collected output RDD.

The procedure of Phase 2 is pretty much the same as the previous one. The only difference is in the data carried by the tuples in input and output RDD and the mapping function.

$$input\ tuple\ B : (i,\ (Sparse\ representation\ of\ i-th\ column\ of\ R))$$

$$output\ tuple\ B : (i,\ (Dense\ vector\ of\ I-th\ row\ of\ updated\ Q))$$

2.2. Implementation of Fast eALS algorithm

The Fast eALS algorithm was coded in the Fast_eALS class. Its members are listed below:

Member variables:

1. **K**, number of latent factors
2. **λ**, ridge regularization term
3. **M**, number of users
4. **N**, number of items
5. **localP**, latent factors for users, a numpy array with size of (**M**, **K**)
6. **localQ**, latent factors for items, a numpy array with size of (**N**, **K**)
7. **localC**, confidence of missing entries, a numpy array with size of (**N**,)

Note that all member variables are stored locally in the RAM of the driver node. As the number K is typically smaller than1000, the largest array localP takes up approximately a few GiB, which is totally affordable by the driver node. Two latent factor matrices are volatile and are updated alternatively in each iteration, while the other member variables remain constant.

Member functions:

1. **constructor(M, N, K, lmbd),** initializes the object by specifying 4 constant scalars M, N, K and λ.
2. **init_PQ()**, randomly initializes local matrices P and Q using standard normal distribution.
3. **init_C(fi, sum_f, c0, alpha)**, takes as input the popularity vector f and calculates the confidence vector c, which is given by $c_i = c_0 \frac{f_i^{\alpha}}{\sum_{j=1}^{N} f_j^{\alpha}}$
4. **predict(u, i)**, returns the predicted score that user u may give to the item i. Since the entire $\widehat{R}$ matrix is too large to be reconstructed locally, only a single element $\widehat{r}_{ui}$ can be requested and returned.
5. **fit(R, niter)**, the core of the Fast eALS algorithm, takes as input the interaction matrix R and updates latent factor matrices **P**&**Q** iteratively. The whole procedure is demonstrated in **Figure 3**. First of all, all variables that will be used in parallelized

functions UpdateP() and UpdateQ() are broadcasted once, including K, λ, **C**, **P** and **Q**. There are two phases in one iteration, as described before. The first phase is to update latent factor matrix **P** while the second one is to update matrix **Q.** In the first phase, matrix **SQ** (with size of K*K) is calculated locally and then broadcasted together with the matrix **Q**. An RDD of all rows of the interaction matrix R is obtained by converting it into an IndexRowMatrix, each tuple (input tuple A) containing both the row index u and the sparse representation of u-th row of R. A mapping function UpdateP() is applied to each of the tuples, converting it into output tuple (output tuple A) which contains both row index u and the u-th row of updated matrix **P**. In the second phase, matrix **SP** (with size of K*K) is calculated and broadcasted together with the matrix **P**. The transposed matrix **R** is then converted into an IndexRowMatrix to obtain the RDD of all columns of **R**. The function UpdateQ() is applied on each tuple (input tuple B), resulting in output tuples (output tuple B) that contain column index i and the i-th row of updated matrix **Q**. The mapping function UpdateP(), UpdateQ(), as well as the calculation of training RMSE are described in **sections 2.3**.

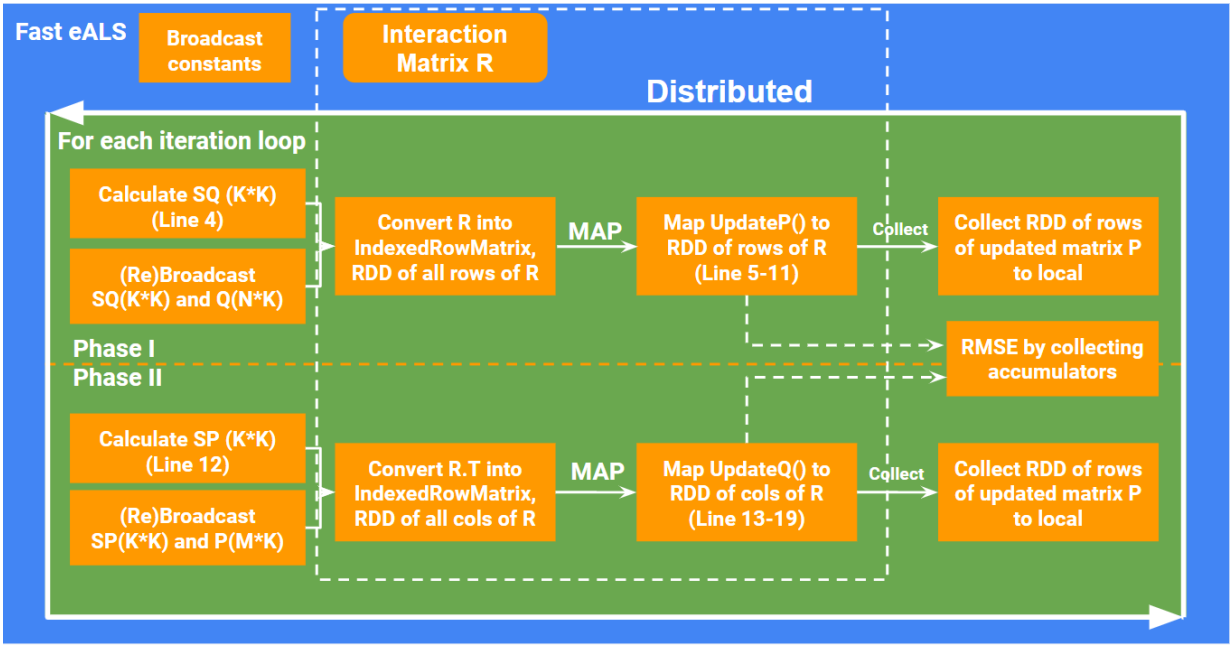


Figure 3. Pipeline of Fast eALS.fit() method

2.3. Mapping function UpdateP and UpdateQ

The two nested loops (**line 5** to **line 11 and line 13** to **line 19** in **Algorithm 1**) are aimed at updating parameters in matrix P and matrix Q, respectively. As the two nested loops are of exactly the same logic, we just describe the details of the first loop here.

In each iteration of the inner loop (from **line 6** to **line 10** in **Algorithm 1**), a temporary local variable $\hat{r}_{ui}^{f}$ is generated from shared volatile variable $\hat{r}_{ui}$ and a single element p_{uf} the matrix **P** is derived from **Equation 12** accordingly.

In our parallelized implementation, a worker executes a single iteration of the outer loop (loop for each user) individually, by looping for each latent feature k and calculating p_{uf} in sequential order. The total number of parallel tasks of this stage is **M** + **N**. The inner level of the first (from **line 6** to **line 10** in **Algorithm 1**) and the second (from **line 14** to **line 18** in **Algorithm 1**) nest loops are coded in function UpdateP() and UpdateQ() respectively. We made the most of Numpy acceleration by using numpy-native matrix operators and avoided any unnecessary for-loops in both functions. Only one necessary “loop for K” remains as they must be executed in sequential order (discussed in the beginning of this section).

All variables that are needed in the function UpdateP() and how the work gets access to these essential variables are listed below:

1. Constant scalars: number of users **M**, number of items **N**, number of latent features **K**, ridge regularization term **λ**. Sharing these constants is convenient, by either broadcasting or passing arguments during function call.
2. Constant matrices: confidence vector **c**. Numpy array objects are too large to be all transmitted via function, but it’s totally feasible via broadcast. As **c** is constant once constructed, we just broadcasted it once.
3. Volatile matrices: latent factor matrix **P**&**Q**, cached matrix **SP**&**SQ**. Like the constant matrices, volatile matrices can only be transmitted via broadcast. On the other hand, they should be re-broadcasted upon modification. Re-broadcasting huge objects such as **P** in each iteration significantly impairs the performance of the algorithm, as described in **section 3**.
4. RDD objects: u-th row of interaction matrix R_{u*}, u-th row of latent factor matrix P_{u*}. Tuples of RDD are transmitted into the mapping function via first function argument and can be transmitted out of the mapping function by returning. In the beginning of the function, the input tuple is parsed into the index u and the sparse vector R_{u*}. In the end of the function, the product (u-th row of latent factor matrix P_{u*}) is wrapped into the output tuple and will be collected by the driver node.
5. Temporary variables: predicted interaction score between user u and item i $\widehat{r}_{ui}$, predicted interaction score between user u and item i with latent factor f excluding $\widehat{r}^{f}_{ui}$. Both $\widehat{r}_{ui}$ and $\widehat{r}^{f}_{ui}$ can be derived from other variables that have been already transmitted to the function. We can either compute them in the driver node and transmit them to worker nodes or just re-evaluate them on demand in worker nodes. In our current implementation, both the input (to be fed to the mapping function) and output (to be collected after mapping function) RDDs contain exactly one key and one value (see tuples definition in **section 2.1**), which makes an simple but elegant “map-collect” pipeline available. If we want to transmit indexed vector $\widehat{r}_{ui}$ as well, in order to create an input RDD with tuples in the form of (u, R_{u*}, $\widehat{r}_{ui}$), a join operation is needed. The shuffle operation of the giant RDD object among all workers can be heavy and slow.

3. Results

3.1. Scalability test and cluster configuration

Our map-reduce implementation was developed online on Databricks Community platform. The scalability test was also conducted on Databricks by setting up AWS EC2 cluster managed by Databricks DBU. The Databricks runtime version is 6.4 (Apache Spark 2.4.5, Scala 2.11). Both the driver and the workers are AWS EC2 r4.2xlarge instances, which are equipped with 8 cores and 61 GiB memory.

We were concerned about three dimensions of scalability:

1. Scale of dataset, including # of reviews $|R|$, # of users M and # of items N. The authors of the eALS algorithm declared that the time complexity is $O((M+N)K^2+|R|K)$. So ideally the time consumption should be directly proportional to the scale of the training set. Any parallelization overhead can lead to non-proportionality. In our test, we divided the dataset into several subsets, as described in **Table 2**.

Table 2. Statistics of subsets of the original dataset

Dataset scale	# Reviews R	# Users M	# Items N	Sparsity	% of (M+N)
Full(1.6M)	6,396,215	1,627,156	192,606	99.99796%	100.00%
800K	1,312,331	800,000	80,000	99.99795%	20.42%
400K	326,850	400,000	40,000	99.99796%	5.11%
200K	82,084	200,000	20,000	99.99795%	1.28%
Tiny(100K)	22,247	100,000	10,000	99.99778%	0.32%

2. # of latent factor K, an user-defined parameter, directly controls the dimension of latent factor matrices **P** and **Q**. According to the declared time complexity, the time consumption should be marginally proportional to K^2. In the test, we chose K ranging from 32 to 192.
3. # of workers in parallel, or # of cores in the cluster. the time consumption should be inversely proportional to the # of workers if parallelization overhead is not taken into account. In our cluster, the # of EC2 instances ranged from 1 to 3, with each instance possessing 8 cores. So # of workers ranged from 8 to 24.

3.2. Results of Scalability test

The results of the scalability test of our eALS implementation is in **Appendix Table 1**.

As shown in the table, we used different combinations of scale, cores, K values to test the performance and scalability of the Fast eALS algorithm. We had some interesting findings through comparison:

1. **The larger the dataset, the higher the RMSE**. When holding other factors fixed, we noticed that datasets of larger scale tend to have higher RMSE, probably because larger datasets contain more noises and outliers which make the training and fitting less accurate. When using the full scale data, the best result we got is 1.59 whereas it was only 0.43 for 100K data;
2. **The RMSE didn't change with the number of cores**. This is straightforward because the number of cores only determined the number of tasks executed at the same time but didn't impact the model parameters;
3. **The RMSE decreased as K increased, but not in a linear way**. As expected, a higher K value introduced more features to describe the users and items, thus making the prediction more accurate. However, the decrease in RMSE was not in linear proportion to the increase in K (marginally decreasing), probably due to the fact that as more features were added in, the newly added ones were more likely to contain redundant information thus contributing less to the prediction accuracy. For example, when performed on the full-scale data using 24 cores, the RMSE dropped from 2.51 to 2.2 as K increased from 32 to 64 (-0.31/32=-0.01) and then to 1.84 as K increased to 128 (-0.36/64=-0.006).
4. **The time consumption per iteration changed with the dataset size, the number of available cores and the value of K,** as demonstrated in Appendix **Figure 2:**
    a. When the number of cores increased, the time consumption per iteration of the worker nodes decreased while that of the driver nodes remained relatively constant. That being said, the decrease in the time consumption per iteration of the workers nodes is not linearly proportional to the increase in cores probably because of the overhead costs (network communication, etc.). It is, however, linearly proportional to the inverse of the number of cores;
    b. When holding other factors fixed, the larger the dataset, the more time it requires to finish a single iteration, for both driver nodes and worker nodes. From this graph we can observe that the dataset size has an impact on time consumption not only by enlarging the theoretical time complexity. We assume that the dataset size makes impact also by increasing overhead of broadcasting and local tasks.
    c. When holding other factors fixed, the more latent factors there are (the larger the value of K), the more it requires to finish a single iteration, for both driver nodes and worker nodes, and the relationship is close to linear.
5. **The time consumption of the UpdateP() and UpdateQ() functions didn't change with cores but only with K**. When more cores were added, the time consumption of the two update functions didn't change much; but when K increased, the time consumption also increased accordingly and in a marginally increasing way. For example, when performed on the full-scale data using 24 cores, the time consumption of UpdateP() increased from 19 to 34 then to 84 and 150 when K increased from 32 to 64 then to 128 and 192 (0.47/0.78/1.03).

3.3. Parallelization Overhead Analysis

We propose that the following 3 factors may result in imperfect scalability and therefore will provide statistical analysis to address these proposed factors.

3.3.1. Overhead caused by imperfect implementation of mapping functions

The performance of mapping function UpdateP() and UpdateQ() is critical to the overall scalability since they will be executed M and N times in each iteration, respectively. Given that both M and N are large numbers (10K ~ 1M), any imperfection of the codelines in the mapping functions can lead to huge waste of computation resources. We reversioned both mapping functions 4 times to make the most of numpy acceleration and removed all unnecessary inefficient for-loops. All computations in the algorithm were rewritten into matrix form and therefore coded with powerful numpy-native matrix operators.
From **Appendix Figure 1.**, we can conclude that both mapping functions scale perfectly with increasing theoretical time complexity. Function call overhead is inevitable when K is relatively small. The CPU needs a fixed amount of time to manage the stack and restore the context during every function call-return cycle. Note that the marginal execution time of mapping functions is the median execution time of worker tasks reported by pyspark UI, with broadcasting overhead excluded.

3.3.2. Overhead caused by broadcasting huge objects

From the execution time of worker tasks reported by pyspark UI, we observed a huge discrepancy between the initial task and the successive tasks taken by a single worker. we assumed that the observed discrepancy was caused by re-broadcasting huge objects such as matrix P (with size M*K ~ 1GiB) and matrix Q (with size N*K ~ 100MB) before executing UpdateQ() and UpdateP() for the first time respectively. The cluster needs to copy the object broadcasted from the driver to workers before everything begins, and it takes a considerable amount of time to copy a huge amount of data via network.
In **Appendix Table 2.** We demonstrated the broadcast overhead (in seconds) by simply calculating the difference between the execution time of the initial task and that of the successive tasks taken by a single worker. We observed that the broadcast overhead is roughly proportional to K and irrelevant to # of cores, which fits our presumption well.

3.3.3. Overhead caused by non-parallelizable tasks

In one single training iteration (**Figure 3.**), there are also some non-parallelizable tasks that have to be done locally by the driver. Two of these tasks may be time-consuming as a large matrix operation is involved:

1. To calculate cached matrix **SQ** and **SP** before invoking UpdateP() and UpdateQ() respectively.
2. To parse the output tuples and to update the matrix **P** and **Q** after returning from UpdateP() and UpdateQ() respectively.

During the execution of local tasks, all workers are idle. The execution of local tasks usually consumes a fixed period of time, irrelevant to the # of cores. As demonstrated in **Appendix**

Table 3, the absolute value of time consumption of the local tasks was fixed but the proportion was increasing when # of cores increased. It will lead to imperfect scalability when the number of cores is relatively large, but it's not the case in our tests.

4. Discussion

In the previous section we analyzed the performance of eALS algorithm from the perspective of scalability, which is the main objective of this report. Besides, we've also found some interesting issues to discuss. Due to the limitation of the budget of our cluster, we didn't investigate these issues in a very careful way.

4.1. Algorithm

In the scalability test we also monitored the change of RMSE over iterations. We observed that the algorithm converged rapidly on the full scale dataset , in 2~3 iterations. But in some cases. the algorithm diverged after the 3rd epoch. Decreasing the size of training data or increasing the number of latent factor K stabilized the algorithm but retarded the convergence. In theory, the algorithm is trying to reach an optimal solution in a non-convex space by jumping to explicit critical points. Thanks to the coordinate-wise method, the algorithm converged rapidly and was able to avoid minor local minima. However, when the dimension of the dataset increases, local minima and saddle points are more likely to disturb the algorithm. As the algorithm is not gradient-based, the RMSE is going to explode as soon as the algorithm decides to jump to a suboptimal critical point.

4.2. Debugging

During the scalability test we've encountered with a fatal error "*OverflowError: cannot serialize a string larger than 4GiB*" that prevented us from increasing # of latent factors K to 256. We investigated this issue and found that broadcasting an object larger than 4GiB is forbidden in pySpark. Once an object is broadcasted, this object will be serialized locally first and sent to all workers via network. Serialization is pretty much like compressing an object into a compact binary data block. The compressed block is then received by workers and deserialized (to restore the object back to its original data structure) in RAM. The largest data block we can send via network is somehow limited to 4GiB by pySpark. Given the fact that 4GiB is also the limit of 32-bit memory addressing, we naturally assume that a byte array indexed by an int32 is used to contain the serialized data. So the largest possible size of the block is $1Byte * 2^{32} = 4GiB$. If broadcasting an object larger than this limit is needed, perhaps we should split them into 4GiB chunks first. Investigation into this issue also led us to another interesting discovery. From the fact that this 4GiB prevented us from increasing K from 192 to 256, we deducted that the dtype of our numpy array **P** and **Q**, initialized by *np.random.normal* function, is float64(double). Training a machine learning model in such a high precision is normally unnecessary and usually a waste of computational resources.

When we were trying to wrap training iterations into a function, we encountered another issue related to broadcasting. Only the broadcast handles that are declared in global scope or passed directly to the mapping function via argument is visible to workers. Declaring broadcast handles in global scope is not an elegant way of coding, and also prevents you

from re-broadcasting them in each iteration. The only way we've found is to declare broadcasting in local scope (Fast_eALS.fit() method) and to send them to the mapping functions via argument passing, which results in an extremely long function signature.

4.3. Future work

We've left many undiscussed topics behind since only a limited amount of cluster budget is available for us to conduct tests. At the early stage of this project, we've tried to run tests locally on a laptop with a tiny subset of data. But we've found that these tests were useless since the algorithm had totally different behaviors with datasets of different scales.

PySpark splits our dataset into 200 partitions by default, regardless of the size of RDD. According to our overhead analysis, we can try to improve the overall performance by tuning the number of partitions. To stabilize the algorithm, we set α to be zero. Thus the confidence vector **c** is uniform. By fine-tuning α we can assign different confidences of missing ratings to be negative for each of the items. In addition, we didn't implement the incremental update for eALS algorithm.

APPENDIX

Appendix Table 1. Summary of Scalability tests

Dataset scale	# Cores	K	RMSE	R	M+N	Theoretical time complexity	time consumption per iteration (seconds)
Full	24	32	2.51	6.40E+06	1.82E+06	2.07E+09	160
		64	2.2	6.40E+06	1.82E+06	7.86E+09	269
		128	1.84	6.40E+06	1.82E+06	3.06E+10	533
		192	1.59	6.40E+06	1.82E+06	6.83E+10	999
Full	16	32	2.54	6.40E+06	1.82E+06	2.07E+09	272
		64	2.21	6.40E+06	1.82E+06	7.86E+09	366
		128	1.86	6.40E+06	1.82E+06	3.06E+10	673
		192	1.59	6.40E+06	1.82E+06	6.83E+10	1231
Full	8	32	2.51	6.40E+06	1.82E+06	2.07E+09	396
		64	2.27	6.40E+06	1.82E+06	7.86E+09	666
		128	1.85	6.40E+06	1.82E+06	3.06E+10	1260
100K	16	32	0.85	2.22E+04	1.10E+05	1.13E+08	7
		64	0.56	2.22E+04	1.10E+05	4.52E+08	10
		128	0.43	2.22E+04	1.10E+05	1.81E+09	16
200K	16	32	1.38	8.21E+04	2.20E+05	2.28E+08	14
		64	0.97	8.21E+04	2.20E+05	9.06E+08	21
		128	0.74	8.21E+04	2.20E+05	3.61E+09	36
400K	16	32	1.78	3.27E+05	4.40E+05	4.61E+08	30
		64	1.39	3.27E+05	4.40E+05	1.82E+09	50
		128	1.06	3.27E+05	4.40E+05	7.25E+09	89
800K	16	32	2.14	1.31E+06	8.80E+05	9.43E+08	73
		64	1.73	1.31E+06	8.80E+05	3.69E+09	124
		128	1.33	1.31E+06	8.80E+05	1.46E+10	235

Appendix Table 2. Overhead caused by broadcasting objects

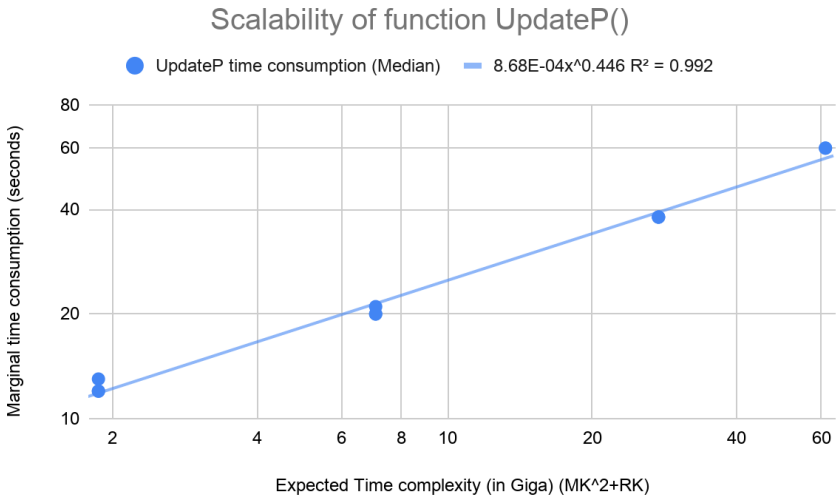
Dataset scale	# Cores	K	UpdateP time consumption (Initial partition)	UpdateP time consumption (Median)	Broadcast overhead P (seconds)	UpdateQ time consumption (Initial partition)	UpdateQ time consumption (Median)	Broadcast overhead Q (seconds)
Full	24	32	19	12	7	13	2	11
		64	34	21	13	26	3	23

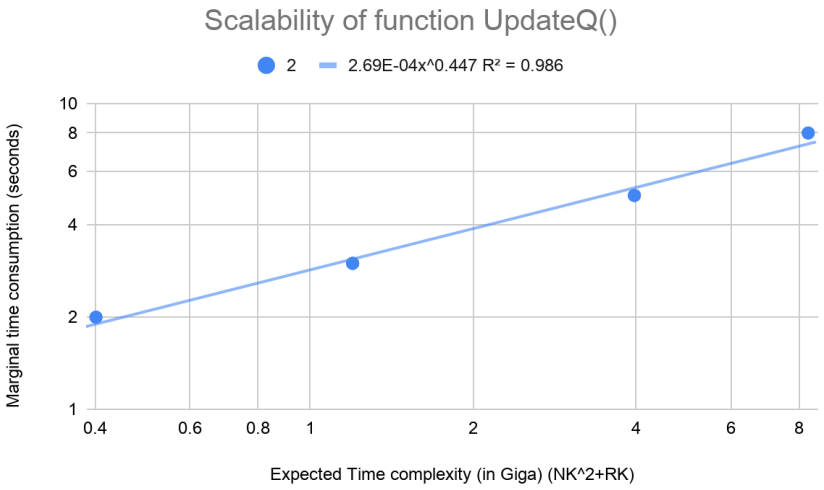
		128	84	38	46	52	5	47
		192	150	60	90	114	8	106
Full	16	32	26	13	13	14	2	12
		64	33	20	13	25	3	22
		128	84	38	46	52	5	47
		192	156	60	96	102	8	94
Full	8	32	19	12	7	13	2	11
		64	33	20	13	26	3	23
		128	90	38	52	56	5	51

Appendix Table 3. Overhead caused by local tasks

Dataset scale	# Cores	K	time consumption per iteration	time consumption per iteration (driver)	% of local tasks per iteration
Full	24	32	160	5	3.13%
		64	269	8	2.97%
		128	533	18	3.38%
		192	999	32	3.20%
Full	16	32	272	4	1.47%
		64	366	8	2.19%
		128	673	18	2.67%
		192	1231	35	2.84%
Full	8	32	396	5	1.26%
		64	666	8	1.20%
		128	1260	18	1.43%

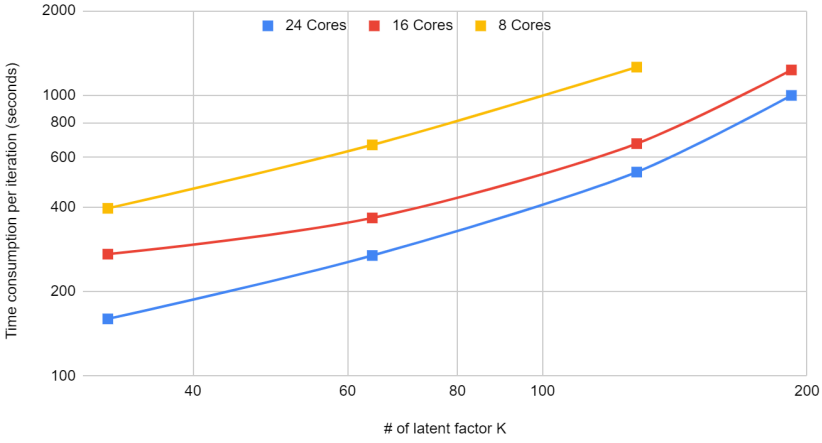
Appendix Figure 1. Scalability of mapping functions



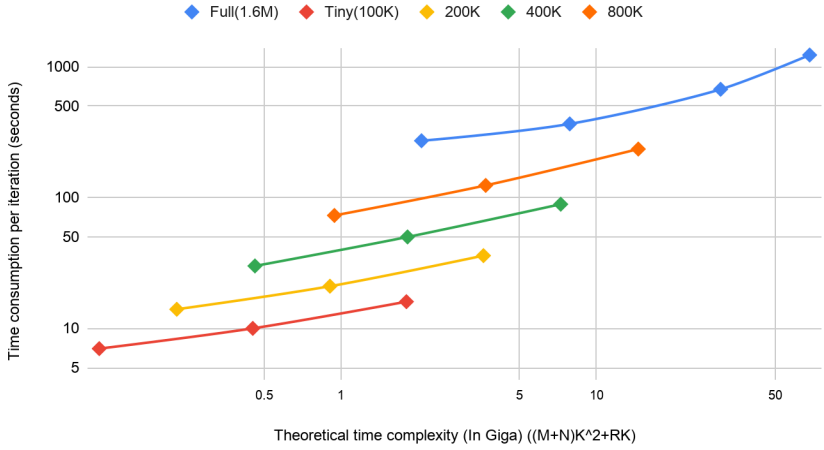


Appendix Figure 2. Scalability results related to K, # Cores and data size

Scalability(A): Time consumption versus K on different sizes of cluster



Scalability(B): Time consumption versus complexity on different sizes of dataset



Scalability(C): Time consumption versus inverse of # cores on different K