Analysis of the common reccommendation systems with the common frameworks: Spark and Flink

Final Report for the BigData project

Mirko Morandi
University of Trento
176043
mirko.morandi@unitn.it

Zhiheng Xu
University of Trento
174222
zhiheng.xu@unin.it

ABSTRACT

In this paper we provide an extensive analysis of the actual state of the art of recommendation systems.
Collaborative Filtering is the current buzzword in the world of recommendations, came to notoriety after the Netflix Prize challenge. In this paper we aim to analyze the current implementations of two different algorithms used for Collaborative Filtering: **Alternating Least Squares** (ALS) and **Stochastic Gradient Descent** (SGD) in combination with the common frameworks `Spark`.

Keywords

Spark; CF; Collaborative Filtering; ALS; SGD; Scala

1. INTRODUCTION

Recommender systems are now trending due to the overwhelming availability of data. These systems have the ability to discover hidden relationships between users and items, and use these patterns to improve the user's taste prediction. Researchers discovered a "neighbourhood" of users with a similar taste which can be revealed by their previous actions: both implicit and explicit. `Collaborative filtering` is by far the most common approach adapted also by some of the biggest companies in the IT sector such as: **Amazon**, **Facebook** and **Netflix**. Although it's massive presence in the market, `CF` is not the only approach available for a recommendation system, but it is actually the successor of `Content-Based filtering`. The latter aims to profile a user searching the correlation with the item's peculiarity. By item peculiarity we refer to its implicit and explicit characteristics, for example a song's genre, subgenre, writer, composer, year of composition, beats per second etc. The problem with this approach lays in the difficult of retrieving all the necessary information, which sometimes are not even available or discloable. Furthermore with the raise of the Big Data paradigm some frameworks started to grow from the academic world to the Apache Foundation: **Flink** and **Spark**. Those frameworks can be seen an extension of the Hadoop ecosystem, and both of them have their own pros and cons which will be briefly analyzed further in this paper.

2. COLLABORATIVE FILTERING

The paper is structured as follow: description in more details of `Collaborative Filtering` with it's problems, what are the most common algorithms used with `CF` and a brief introduction to both **Flink** and **Spark**.

2.1 Collaborative Filtering Approaches

Collaborative Filtering can be subsenquently defined in two different approaches:

2.1.1 *Memory-based Collaborative Filtering*

In `memory-based CF` uses user ratings to compute similarity between user and items and subsequently make a recommendation. Usually this approach involves "neighboring" algorithms such as **K-Nearest Neighbours** to build relationships between users. The similarity between two users is calculated using the **cosine similarity**.
Cosine similarity is a measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them.[4]

$$\cos(\mathbf{t},\mathbf{e}) = \frac{\mathbf{te}}{\|\mathbf{t}\|\|\mathbf{e}\|} = \frac{\sum_{i=1}^{n} \mathbf{t}_i \mathbf{e}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{t}_i)^2}\sqrt{\sum_{i=1}^{n} (\mathbf{e}_i)^2}} \qquad (1)$$

The recommendation is made by finding the top K similar users and aggregate their user-item matrices to find the appropriate recommendation. The typical problem of this approach is the difficult with scaling when the data gets bigger. Due to the `Big Data` paradigm expansion this approach has been deprecated favoring the following approach.

2.1.2 *Model-based Collaborative Filtering*

The most common approach to `CF` is through the factorization of a very big and sparse matrix.[3] For example during the Netflix Prize at the participants were given a matrix of 8.5 billions of ratings, of which only 100 millions were non zero values. *Model-based CF* uses machine-learning and data mining algorithms to uncover the latent factor model between users and items to predict the missing ratings.**latent factor models** are hidden relationships between users and items hardly discoverable in the original data; usually they may for example denote the quantity of action in a movie

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Table 1: Example of a sparse user-item ratings matrix

items/users	U1	U2	U3	U4	U5
I1	5	3	-	1	3
I2	4	-	2	1	-
I3	2	2	-	5	-
I4	4	3	-	4	2
I5	-	5	5	4	5

or the complexity of the characters.These vectors are then used to create the missing values in the user-items matrix.

Furthermore the model-based content filtering can be expanded in two distinct sections: *user and item based content filtering* depending on the priority given to the prediction.

2.2 Related problems

2.2.1 Cold start problem

Due to the nature of CF, the system needs a huge amount of data in order to produce a reliable prediction. But what happens if our system hasn't collected any or not enough information yet? This problem is called **cold start** and can be tackled with some advanced machine learning solutions called *active learning.*

2.2.2 Shilling Attacks

CF can be exploited to perturbate its prediction system with a technique called *shilling attacks*. This happens when the input system (e.g. ratings) are given in the correct way trying to alter the recommendations to the one favoured by the attacker. It has also been noticed that these kind of attacks affect more user-based CF algorithms instead of item-based.[5]

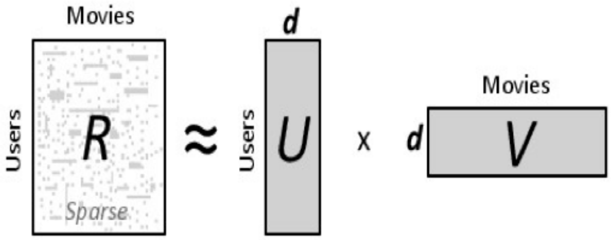
2.2.3 Sparsity

In the era of the *web 2.0* the matrices who compose the datasets are usually very sparse due to the typical proportion for which $nUsers \ll nItems$. As said previously the matrix which was given had only 100M ratings out of 8.5 billions of records. This problem is solved using matrix factorization.

3. MATRIX FACTORIZATION

During the last decade a huge effort has been applied to solve the problem of big datasets with an incredible amount of data. Let denote $R = r_{ij}$ denote a user-movie where each element r_{ij} represents a rating given by a user i to an item j with a value from 0 to 5; where 0 means non rated and 1 to 5 a rating ranging relatively from *very poor* to *awesome*. Let also define m the number of users and n the number of movies in the system. The problem of *recommender systems* is to predict the missing values of R using the known ratings.

The concept behind *matrix factorization* is to find two matrices V, W relatively $\boldsymbol{m}$ x $\boldsymbol{p}$,$\boldsymbol{n}$ x $\boldsymbol{p}$. which product can approximate a much bigger matrix R of dimensions $\boldsymbol{m}$ x $\boldsymbol{n}$. $R \approx V * W^T$



The process consists in a low-rank approximation of the user-item matrix using for both users and items some feature vectors which are used to model the prediction with a inner vector product of the selected user and item.
Ideally r_{ij} should correspond to the predicted rating but in reality the value will be different. Therefore we need a loss function to determine the difference between **real value** and **predicted value**. In our paper we are going to use the **Root Mean Square Error**,

$$\sqrt{\frac{1}{N}\sum_{i,j}^{N}(pr_{ij} - r_{ij})^2} \qquad (2)$$

Where respectively *prij* is the prediction for the rating given by the user i for an item j and *rij* is the real rating for the former user and item. The prediction is obtained with the dot product of the user-item vectors,(*ui*,*vj*)

$$prediction_{ij} = \bar{u}_i * \bar{v}_j \qquad (3)$$

The low rank approximation problem is formulated as follows to learn the factor vectors (ui,vj) [6]

$$(u_i, v_j) = \min_{u,v} \sum_{(u,i)\in K} (pr_{u,v} - r_{u,v})^2 \qquad (4)$$

However solving the approximation with the former approach leads to the overfitting of data. Overfitting means that our model is learning through the noise in the training data instead of the underlying relations for which we expect to work. Although the performance will continue improving with the training data it will get worse on unseen data. This usually happens when we are using a too complex model or we have too many free parameters, and the latter is our case. The solution is to apply regularization penzaling the magnitude of the feature vectors proportionally with a constant factor λ obtaining the following and well known formula for low-rank approximation:

$$(u_i, v_j) = \min_{u,v} \sum_{(u,i)\in K} (pr_{u,v} - r_{u,v})^2 + \lambda(u_i^2 + v_j^2) \qquad (5)$$

3.1 Approaches

There are mainly two approaches to minimize equation(5) and are known as:
Alternating Least Square and **Stochastic Gradient Descent**.

3.2 Stochastic Gradient Descent

Stochastic Gradient Descent is a gradient descent optimization used to minimize a loss function, in our case equation (5). The goal of *Stochastic Gradient Descent* is to find a value $\theta^* \in R^k (k \geq 1)$ that minimizes a given loss $L(\theta)$. The algorithm makes use of noisy observations $\bar{L}'(\theta)$ of $L'(\theta)$, the function's gradient with respect to θ. Starti with some initial value θ_0, SGD refines the parameter value by iterating the stochastic difference equation[7]

$$\theta_{n+1} = \theta_n - \epsilon_n \bar{L}'(\theta_n) \tag{6}$$

where n denotes the step number and ϵ_n is a sequence of decreasing step size.

3.2.1 SGD with Matrix Factorization

In matrix factorization we set $\theta = (V, W)$ and the loss function can be written as $Q(w) = \sum Q(u, i)$. Where w is an approximation obtained iterating through every single sample until a minimum is reached with the following step: $w = w - aplha * d\frac{d}{w}(Q_i(w))$ In our case $Q(u, i)$ is represented by equation(5), and is the equation we want to minimize. The iterative version of the algorithm can be sketched this way:[8]

Algorithm 1 Matrix Factorization with SGD

```
Require:
    R is the user-item matrix,
    V and W are the factor vectors initialized with values from 0.0 to 1.0,
    $\alpha$ is the learning rate,
    $\lambda$ the magnitude reduction,
    K is the number of iterations
    F is the number of features
shuffle(Ratings)
for $i$ to $K$ do
    for user $u$ and item $i$ with a rating $r \in R[u, i]$ do
        $predictedRating = V[u] * W[i]^t$
        $error = R[u, i] - predictedRating$
        for each feature $f \in F$ do
            $V[u, f] = V[u, f] + \alpha * (error * W[i, f] - \lambda * V[u, f])$
            $W[i, f] = V[u, f] + \alpha * (error * V[i, f] - \lambda * W[u, f])$
        end for
    end for
end for
```

Although this algorithm is used to optimizie equation(5) it has some scalability issues. These problem are insights of its iterative nature for which all the individual steps depends on each other and make it hard (but not impossible) to distribute it In this paper we will propose a functional implementation of this algorithm, and try to optimize it with a distributed version.

3.3 Alternating Least Square

Because both *ui* and *vj* are unknowns, Equation 5 is not convex. However, if we fix one of the unknowns, the optimization problem becomes quadratic and can be solved optimally. ALS is the techniques that rotate between fixing the ui âĂŹs and fixing the vj âĂŹs. When all ui âĂŹs are fixed, the system recomputes the vj âĂŹs by solving a least-squares problem, and vice versa. This ensures that each step decreases Equation 5 until convergence. In this section, the cost function is defined as,

$$Q(U, V) = \sum_{(i,j) \in K} (pr_{i,j} - r_{i,j})^2 + \lambda(u_i^2 + v_j^2) \tag{7}$$

The detailed description is as following,

Algorithm 2 Matrix Factorization with ALS

```
Require:
    R is the user-item matrix,
    U and V are the factor vectors initialized with values from 0.0 to 1.0,
    $\alpha$ is the learning rate,
    $\lambda$ the magnitude reduction,
    K is the number of iterations
    rmse is expected RMSE
    ri is the ith row of R, rj is the jth column of R
    for $k$ to $K$ or RMSE > rmse do
        fix V, and caculate the partial derivative for ui, and make it equal to 0, we have
        $u_i = (V^T V + \lambda I)^{-1} V^T r_i$
        update all the ui
        then fix U, and caculate the partial derivative for vj, and make it equal to 0, we have
        $v_j = (U^T U + \lambda I)^{-1} U^T r_j$
        update all the vj
end for
```

One obvious advantage of ALS is that the system can easily use parallelization. In ALS, the system computes each ui independently of the other item factors and computes each vj independently of the other user factors.It means that the algorithm can update all the ui and vj in parallel. This gives rise to potentially massive parallelization of the algorithm and improve the efficiency.

3.4 Machine Learning and BigData

Machine learning is ideal for exploiting the opportunities hidden in big data. It delivers on the promise of extracting value from big and disparate data sources with far less reliance on human direction. It is data driven and runs at machine scale. It is well suited to the complexity of dealing with disparate data sources and the huge variety of variables and amounts of data involved. And unlike traditional analysis, machine learning thrives on growing datasets. The more data fed into a machine learning system, the more it can learn and apply the results to higher quality insights. Freed from the limitations of human scale thinking and analysis, machine learning is able to discover and display the patterns buried in the data.[9] Data by itself is useless if there is no way to extract information from it. That's why BigData goes always in couple with machine learning or data mining; these two different but linked branches of computer science aims to extract or predict relevant data from a huge amount of information. *Data Mining* is a field of computer science where algorithms tries to find patterns and correlations analyzing data; there are some subcategories in data mining known as *text mining* and *process mining*. Text mining is used to find correlations between words in a text file, the most famous example is a search engines; search engines needs to find the approriate result matching the keywords analyzing the words in a webpage, the most common approach is **PageRank**. We can also use data mining to

extract relevant information from our log files in order to detect errors, bottlenecs or anomal behaviours (attacks or viruses); this is called **process mining**.
Machine learning is another branch of computer science used to make predictions after ”teaching” the machine how to learn from data. The typical approach is to use a dataset called *training data* (usually 80% of the dataset) to build a model. The training phase is where our algorithm makes predictions using the data and compute the error relative to the real value, and subsequently improve the predictions until convergence (e.g. when reaching a threshold for which the error is acceptable). After reaching the convergence threshold we have to test our data with unknown data (usually the remaining 20% of the dataset) and then compute the predictions on the latter. Typically the error is slightly higher with the test dataset (if we have the same error it means we have a wrong model), otherwise if the difference is significant it means that we are *overfitting* the model. Overfitting means that we are training our model wrongly and it does not learn from the hidden correlations between values but instead from the noise relate to that.

4. TECHNOLOGIES AND CHALLANGES

Up to now we have described what is the so-called state of the art in the reserach enviroment. Our challange in this paper will be to develop, where needed, and compare the two most common approaches to matrix factorization using the *Spark* framework. There’s already an implemented version of the *Alternating Least Square* with scala, which we’ll use to do our tests. Our goal will be to implement the *Stochastic Gradient Descent* starting from the naive iterative version and optimize it aiming to reach a distributed version. We have also choose to user **Scala**, a functional programming language to develop the SGD factorization; functional programming languages seems to be better involved in the bigdata paradigm due to their ability to facilitate parallelization. Last but not least, we have used the various movielens datasets with different sizes to test our examples.

4.1 Scala

Scala, acronym of ”Scalable Language”, is an object-oriented functional programming language used for a large variety of tasks: from scripting to large mission critical systems (e.g. Twitter and Linkedin). Functional programming languages are well known in the bigdata environment due to one of their preculiarities: **The pure functions**. Pure functions are like mathematical functions, the return value depends only on the input value; meaning the function does not have any *side effect*. This is very important when dealing with large datasets because it’s the peculiarity of functional languages that improves parallelization. The requirement for scaling horizontally is to have many independent tasks running in parallel, which goes perfectly with the latter peculiarity of functional languages.

4.2 Spark

4.3 MovieLens Dataset Structure

In our project we used mostly two datasets from Movielens: the 1 million ratings and the full dataset.

4.4 The machines

Table 2: MovieLens datasets

-	**Ratings**	**Users**	**Movies**
1 Million dataset	1.000.000	6.000	4.000
Full dataset	22.000.000	240.000	33.000

Table 3: Computer

OS	**#CPU**	**#Threads**	**RAM**
Mac OS X	2	4	8 GB
Ubuntu 14.04 remote	4	8	8 GB

During our test we used different machines for testing and for production (in other words real external server). For local testing we used

4.5 Libraries

5. IMPLEMENTATION

The first step of our reserach was to develop a ”naive” version of SGD using the various iterations with no synchronization. We also have decided to start using the smalles dataset from movielens to understand how long it will take to complete. The decision was wise because on the first run took around 8.7 hours to complete with a dataser of 1 million of ratings. The overall problem was already introduced in the previous chapters of this paper and it’s the nature of this algorithm to not scale well with big datasets. Moreover if we consider our example and using $nIterations = 25$, $nFeatures = 20$ with $nUsers = 6000$ and $nMovies = 4000$ we would have to compute nIterations * nUsers multiplied by the ratedmovies the dot product of the user item factor vectors, and update the values for nFeatures times. We can summarize the cost of our iteration as $O(nIterations * nUsers * ratedMovies * nFeatures)$. Since the number of users is typically the highest value this doesn’t scale well with big datasets where the number of users reaches very high numbers and the number of computations becomes enormous. Altough the problem of scaling is relative to the algorithm we found out that some commonly used data structures and operations we used had a high impact on the performance.

5.1 The algorithm

In our implementation we used a variant of the simple SGD called biased SGD which considers also some bias related to users and items. Biases are really important in our example, because ratings are personal driven decisions makes them differ from person to person. Let’s say that the overall average is 3.6 (that’s the real average in our dataset) and Start Wars is a movie above the average which means it tends to be rated 0.5 more than other films. On the other hand we have a critical user Bob, who tends to give lower than average ratings to movies around 0.3,in our case the starting point for the prediction would be (3.6 + 0.5 - 0.3) = 3.8. The predicted rating will be
predictedRating = μ + *userbias* + *itembias* + *vectorFactorProduct*

Epoch Loop The first step of the algorithm is to iterate a given number of epochs and update the users. This is the basic loop

User-Item Loop The user-item loop is executed each time inside the former loop. This is a nested loop which goes through all the user and for each of them all the items for which they do have a non-zero rating. For each of this couple of user-items we first predict the value for the item and compute the relative error as follows: $error = predictedValue - realValue$. The value is predicted with the dot product of the factor vectors associated to the relative user and item. Here we also update the biases related to both user and items with the following formula $bias = bias + alpha * (err - biasRegulator * preventOverfitting * bias)$. This allows us to adapt the user bias for any given rating; alpha is a float value, usually around 0.5 used to help the system's precision in the bias computation. PreventOverfitting is another float value used to deacrease the chance of overfitting the data, in our example is always around 0.1. It has been already mentioned during this paper under the name of λ, used to reduce the impact magnitude of each rating.

Feature Loop The feature loop is an iteration for all the features associated to the user and item vector factor; since the number of features is the same for both user and item factors we can solve everything in a single loop.

5.2 The optimization game

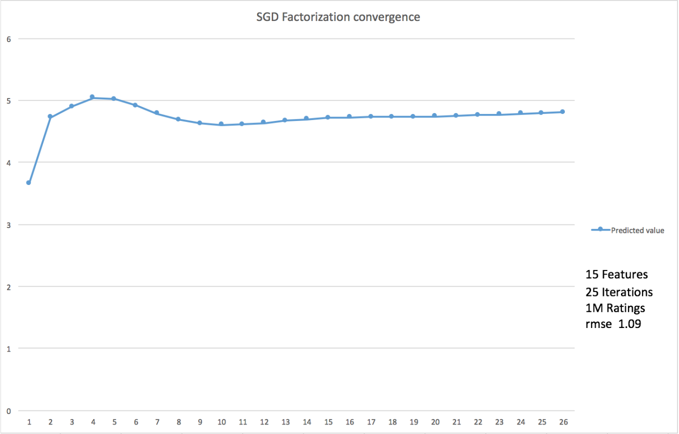
In our first try we made a large use of lists and other structures which didn't perform very well on a large scale. Hereby there's a table of the common mistake we made during the first stage development and how we solved it.

Lists The first mistake we made was to use simple scala *Lists*. Altough their semplicity of use they share a common problem: the cost of retrieving an element. The cost of getting an element from a List is $O(n)$, which means the bigger the array the more time it will take to get an element. Luckily Scala gives also other much better performing structures such as *Arrays* and *Dictionaris*, whose cost of getting an element is rispectively $O(C)$ and $O(eC)$. The first one performs slightly better on a large scale but they do both perform much better than Lists.

Array Altough Arrays are good for retrieving and updating elements they're not so good, on a large scale, for the dot product of vectors. We tried to perform a dot product of two Arrays of 5 millions of random values and two *DenseVectors* from the *breeze* library and the result was astonishing: 4948115 vs 39138 microseconds. We ran the experiment multiple times and the magnitude of the difference was the same even using the *parallel collections* to compute the dot product.

RDD filtering We needed to retrieve at each step the real rating from the dataset and order to compute this operation we filtered directly the distributed Dataset. Even though the dataset is distributed it had a very high cost of information retrieval, a value around 132080 microseconds. We have then decided to cache all the ratings using an *md5 digest* computed using the *userID* a space and the *movieID* to have a unique value (e.g. "2293 345"). This operations has a non negligible cost but after the initial setup the improvement was noticeable: 48 microseconds compared to the previous 132080.

Figure 1: Figure 2



6. RESULTS

The results from our tests proved the correctness of our implementation of SGD altough the performance are slightly worse if compared to the ALS implementation.

1. The first test was run on a local machine with 25 iterations and 15 features.In picture[2] we can see the slow convergence with the typical curve of SGD with a very high peak in the beginning and then a fall until convergence to the real rating, 5.0 in our case. The overall iteration took around 45 minutes to complete with an accettable error: rmse = 1.09 (using a reduced version of the dataset)

2. The second test was a run on the remote server using the extended version of the dataset with 50 iterations and more features, 30. The algorithm with no optimization took 8.7 hours to complete with an rmse = 1.078. The result was good but the running time was out of any acceptable range

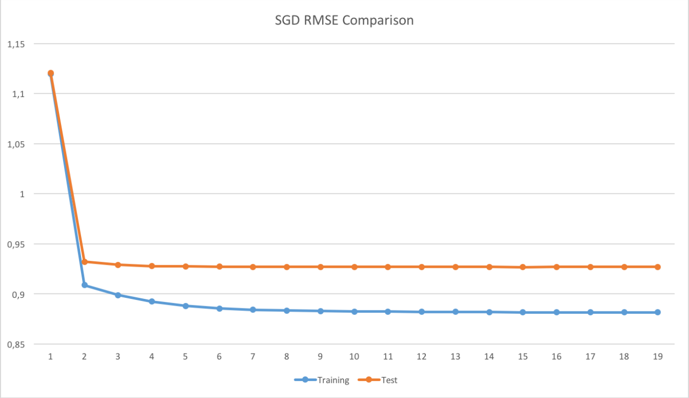
3. The third test was run again on the remote server, this time with an higly optimized version of the algorithm. This time the number of iterations was set to 200 with 30 features. The running time was 1.78 hours with an rmse = 0.92. Compared to the previous test, the running time would be 44 minutes, which is a big improvment.

4. The last test was set up with the parallel version of the algorithm with the same parameters as the previous. The algorithm took 45 minutes to complete the whole process which is an improvement of the 400% compared to the previous version.

6.1 Playing with the RMSE

The most important factor and indicator of the whole algorithm is the *root mean square error*. The rmse indicates the "correctness" of the algorithm, in our case a good value would be from 0.8 to 1.1 at maximum. The typical value for

Figure 2: Figure 3



this dataset is around 0.9[10].
Altough the algorithm was correct in our first batch of test the *rmse* was around 1.29, which is not the best value we could get. We tried to fine tune some parameters, starting with the number of iterations and the number of features, the ones we considered the most relevant in the algorithm. However we were wrong, the error was not decreasing as expected but only of few decimal values. The key to our high value was in the *bias learning rate* or α in the pseudocode, which was too high and was influencing too much the prediction. Using a value around 0.1 we decreased the error of a factor of 32%, with a resulting rmse = 0.9 on the test data and 0.88 on the training data, which is a good result (see figure 3).

6.2 The parallel version

We analyzed the iterative version and in order to proceede in the parallelization we had to find some indipendent parts in our code. We made the assumption that all users may be processes indipendently even though the items will be shared. We solved this problem using concurrency on the items allowing only one of the parallel executions to have it. We have sliced the execution assigning theoretically a thread to each slide until the environment allows us to. In our case the maximum was 1500, but with some fine tuning of the heap and stack parameters one can allocate around 100 thousands of Java threads.[11]

Altough the parallel version of our algorithm needs some locking system due to the concurrency on the items vector when reading and updating it there is a theoretical approach to a more distributed version. The key of this approach is to create a graph with all the interdepences between users and items, where the edges are the correlations. If we have more connected components, those will be fully independant from the others because there will be no concurrency issue between items, therefore we will be able to compute those on different machines and collect the results in the end. The only problem is relative to the possible number of connected components which is by nature of the problem very small, if not only one.

7. CONCLUSIONS

Research not always means finding the new disruptive innovation or the coolest algorithm; most of the times you try you make mistakes and most important of all you *learn.* We learned that very likely Scala is not the best programming language for not parallel algorithms; maybe a C++ solution would have been faster as expected. Maybe Stochastic Gradient Descent is not the best approach for matrix factorization, which would explain why there is no Scala implementation yet instead of the much more spreaded ALS.

Algorithm 3 Matrix Factorization with parallel SGD

```
Require:
    R is the user-item matrix,
    V and W are the factor vectors initialized with values from 0.0 to 1.0,
    $\alpha$ is the learning rate,
    $\lambda$ the magnitude reduction,
    K is the number of iterations
    F is the number of features
shuffle(Ratings)
for $i$ to $K$ do
    run a thread for each user until the maximum number or the limit is reached
    for user $u$ do
        for each item $i$ rated by $u$ do
            lock(item $i$)
            $predictedRating = V[u] * W[i]^t$
            $error = R[u,i] - predictedRating$
            for each feature $f \in F$ do
                $V[u,f] = V[u,f] + \alpha * (error * W[i,f] - \lambda * V[u,f])$
                $W[i,f] = V[u,f] + \alpha * (error * V[i,f] - \lambda * W[u,f])$
            end for
            unlock(item $i$)
        end for
    end for
end for
```

APPENDIX