Recommender system with Funk SVD and Alternating Least Squares matrix factorization

Anže Mur, 63150203
Faculty of Computer and Information Science, Big Data course

Abstract—With the explosive growth of digital information, recommender systems have become an essential part of our lives. They are successfully solving the problem of information overload by filtering important information that can provide enormous value to our businesses and consumers. In this paper, we explore two different parallel approaches for creating recommender systems with two different implemented models - Funk SVD and Alternating Least Squares (ALS). We tested our models locally and on a distributed cluster and we showed that one of our models scales really well. As for the prediction accuracy, we obtained good results since we got an MSE value of 0.8477 with our Funk SVD model and an even better MSE value of 0.8372 with our ALS model while testing on the Amazon Review Data (2018) dataset.

Index Terms—matrix factorization, Funk SVD, ALS, Dask, parallelization

1 Introduction

In the last few decades with the explosive growth of digital information and an increasing number of internet users, recommender systems have become more and more intertwined with our lives. They are successfully solving the problem of information overload by filtering vital pieces of information out of large amounts of user data that consists of his preferences, behavior, and interests. By analyzing this filtered information, the recommender system can predict if some chosen user would like a certain item or not [1].

Recommender systems are useful to both users and service providers and are primarily used in commercial applications. By using a recommender system we can increase revenue and sales of our business, as we can predict the next item the user will want to buy, so we can recommend that item to him and speed up the sale or help the user make that final decision. And by providing the user with a personalized feed of items, he feels more connected to the service and he tends to buy more. Amazon, which is currently considered to be one of the "Big Five" companies and owns the world's largest online marketplace [2] is using recommender systems since 1998 and it is estimated that 30 percent of Amazon.com webpage views come from recommendations. Another great example is Netflix which uses a recommender system to recommend new movies to its users with the goal to extend their watch time on the platform. It is estimated that more than 80 percent of the movies watched on the platform came through recommendations, which translates to more than 1 billion US dollars per year [3].

In the production environments, recommender systems' predictions must be happening fast and in real-time to provide valuable predictions on the spot. The consequence of that is that recommender systems must be highly scalable and must use different techniques and resources to provide that speed and scalability - one of them is parallel computing which we utilized in our work. To achieve that we used Dask with which we successfully parallelized two different matrix factorization algorithms - Funk SVD and Alternating Least Squares (ALS) which are both explained in the forthcoming chapters. To test our implementations and their scalability we set up a distributed cluster on Amazon Web Services (AWS). We also tested the accuracy and scalability of the implementations on the local cluster and evaluated our model in comparison to the Spark MLlib implementations.

2 Collaborative filtering

Since both of our algorithms use collaborative filtering we will shortly discuss the principles of it. Collaborative filtering is a method that recommends new items by identifying other users with similar preferences as the user to whom we want to recommend this item. It is a domain-independent technique and works by building a database (matrix) of item preferences selected by users (user-item matrix). This matrix is used for matching users with similar preferences by calculating similarities between them and then finally to make recommendations. User will be recommended new items - items that he hasn't rated yet, but users similar to him have positively rated them before. We can classify recommendations that are returned from collaborative filtering as a prediction or recommendation. The recommendation is a list of top N items best fitted to the user's preferences and prediction is a numerical value that represents the predicted score of the item [1]. The majority of algorithms using collaborative filtering that is used today operate by generating predictions of the user's item preferences and then produce recommendations based on these obtained predictions [4]. Collaborative filtering techniques are generally divided into two sub-categories: **memory based techniques** and **model based techniques**. In our case, we will be using a model-based technique that assumes a model that describes the user-item interactions and tries to make new predictions based on these discovered relations.

3 Matrix factorization

Matrix factorization is one of the most popular collaborative filtering methods. The method represents a user or an item by projecting them into a shared latent space, with the help of a vector of latent features. Users' interactions on an item are then modeled as the inner product of their latent vectors [5]. The idea behind matrix factorization is to estimate unknown ratings by factorizing the rating matrix (X) into two smaller matrices representing the user (P) and item (Q) latent factors as seen in figure (1). By applying the dot product between these two latent factor matrices we can reconstruct and estimate our rating matrix and we will obtain new ratings for our users in places where there were none.

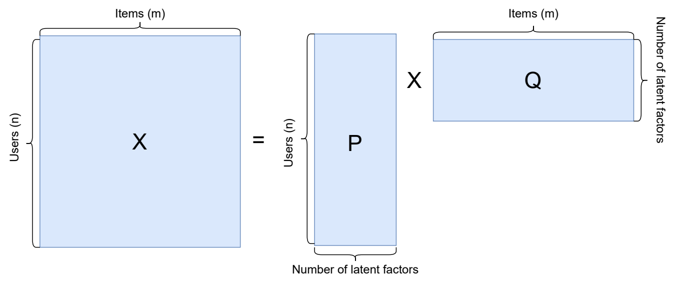

Fig. 1. Visual representation of matrix factorization using user latent factors P and item latent factors Q.

4 Data & data preprocessing

As the data for the development of our algorithms, we used the Amazon Review Data (2018) dataset. The data becomes really sparse when building the user-item matrices and to decrease this sparseness and avoid unrated items and unactive users we decided to use a subset of the original data called 5-core. The reason behind the name of this subset is that every user in this dataset provided at least 5 ratings and similarly every item received at least 5 ratings. This increases the accuracy of our recommendations since collaborative filtering methods need enough data to provide reliable recommendations [6].

Our dataset consist of four different columns: **item, user, rating, time**. The first thing that we did in light of data cleaning is that we removed the duplicates from our dataset. But that didn't really solve our problem since there were still some duplicates in terms of user and item columns - that means that the same user-provided more than just one review for some item, and that can cause us some troubles while building the user-item matrices. To solve this issue we decided to use only the last provided review from the same user as it is the most relevant in our opinion. To achieve that we sorted our dataset based on the time of the given review and removed the duplicates with the same user and item value pairs. After that, there was really no reason to keep the information about the time of the given review so we removed the time column from our dataset. The last step of our data preprocessing was to split our data into training (70%) and testing (30%) sets.

For the local testing, we decided to use a dataset with $219,155$ rows which have $27,482$ unique users and $10,602$ unique items. This was the maximum that our local machine could handle with limited resources. For the distributed cluster testing we choose a dataset that is almost $800,356$ rows and has $101,247$ unique users and $27,892$ unique items. The larger dataset was transformed into parquet file format and uploaded to Amazon's S3 storage for easier access.

5 Funk SVD

5.1 About Funk SVD

Funk SVD was first proposed by Simon Funk in his blog post in 2006 where he described how he and his team obtain very good results using matrix factorization algorithms in Netflix prize competition [7]. Simply put is it a fast implementation of matrix factorization using a stochastic gradient descend (SGD). The algorithm minimizes the loss function on both user and item latent factors at the same time.

5.2 Implementation

Our initial approach was to use Dask data frames which are composed of many Pandas data frames [8]. This data frame is parallel and can be distributed across many different machines in a cluster as opposed to storing a whole data frame in one machine's memory - in this way we can process much larger amounts of data. The idea behind our implementation was to create one big data frame which structure was identical to our original dataset with some added columns: two separate columns for user and item biases and a desired number of columns for the user and item latent vectors. After the construction of our data frame, we were able to easily perform column-wise operations. The final step was to partition our data frame so the workload is evenly balanced. We selected the optimal partition size of 100 Mb. The solution worked very well in terms of processing larger amounts of data but wasn't really performing as we wanted in terms of computation speed. We discovered that computations weren't really happening in parallel so we abandoned this idea and tried the next more scalable approach.

For our second approach, we decided to use Dask arrays which use NumPy under the hood [9]. Dask arrays allow us to cut up arrays that don't fit in our memory, into many smaller array blocks (NumPy arrays) which can then be processed in parallel using all of our cores on the local machine or on a distributed cluster.

The first thing that we had to do before training was to build our user-item matrix. For that, we had to encode our users and items into embeddings as unique indexes starting from zero. With this, we also obtain unique numbers of users and items. A very big problem in collaborative filtering is the sparsity of user-item matrices because only a subset of all items was rated by users [1]. We partly solved this problem by using the 5-core datasets, but our matrix is still highly sparse. This can cause problems with larger datasets since we will need a lot of memory to store this matrix mostly filled with zeros - useless values. To save memory and to perform more efficient computations we decided to present our matrix in Coordinate List (COO) format [10]. This format stores tuple of information about each non-zero value in the sparse matrix - its column and row indexes for the position and the actual value. So instead of storing all of the information in memory, we keep only the data that really matters to us.

The next step was to chunk this created matrix so that we can utilize the power of Dask's block computing. We specified a chunk size parameter that tells us how big a piece of the user-item matrix should each chunk cover. This parameter is very important since it specifies how our work will be divided and parallelized so its size increases with the size of the user-item matrix. Then we created our chunks by evenly dividing our matrix between them. From these chunks, we created the blocked Dask array and our data was finally ready for computation. The final user-matrix form can be seen in figure (2). We also created a mask of our user-item matrix that includes zeros in places where we have an actual rating. The mask is used while computing the learning error - if we multiply the same-sized array with it we will take into account only previously existing ratings so we can evaluate our results only on the ground truth existing values.

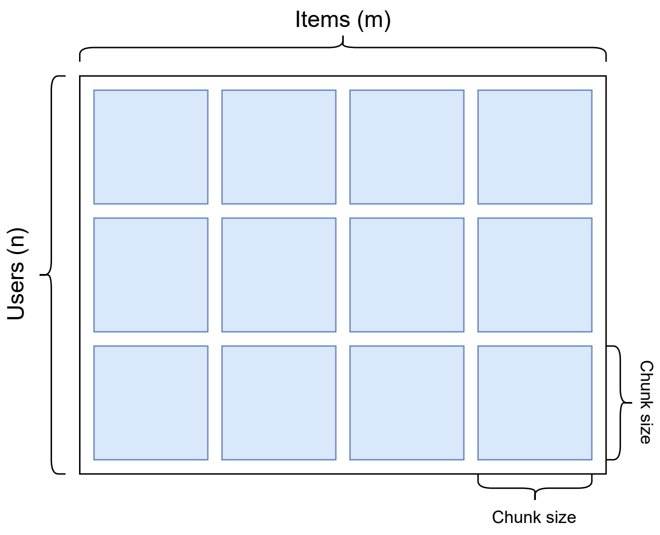

Fig. 2. User-item matrix of dimenison $n \times m$ evenly distributed between chunks of size $chunk\ size \times chunk\ size$.

In the next step, we created user and item biases and latent factors. User and item biases are one-dimensional arrays of zeros with sizes of unique users and items. For the latent factors, we generated random two-dimensional arrays using the normal distribution with an offset of 0.1 around zero. The second dimension of these arrays is determined as an input in our algorithm and represents the number of latent factors that we want to use. Both biases and latent factors arrays are also chunked so we can perform blockwise operations. With this step, we finished with all of the necessary preparations so we can start with the actual training of the model

In each learning iteration or epoch we perfrom stochastic gradient descent (SGD) to optimize our model. We calculate the current estimation of the user-item matrix based on user and item latent factors and biases:

$$X_{est} = mean(X) + bu + bi + P * U$$

where bu represents user biases and bi represents item biases. From this estimation we can then calculate the learning error:

$$ERROR = X - X_{est} * X_{mask}$$

where X_{mask} represents the calculated mask of the user-item matrix. We use this mask to preform the evaluation only on values that are accutally in the user-item matrix. We can then use this error to update the user and item biases:

$$bu = bu + \alpha * (ERROR - \lambda * bu)$$

$$bi = bi + \alpha * (ERROR - \lambda * bi)$$

The last step of iteration is updating the user and item latent factors:

$$P = P + \alpha * (ERROR * Q - \lambda * P)$$

$$Q = Q + \alpha * (ERROR * P - \lambda * Q)$$

where α represents the learning rate and λ regularization factor. All of the operations are performed blockwise with Dask and are nicely parallelized as we can see in figure (3).

The key for the optimal paralelization is that we don't compute the values in every iteration, but we stack the computations together and compute the acutal values after the last iterations. This way we give up the option of early stopping when our model converges but we gain allot of computational speed which in our case is our priority. The right amount of iterations for the optimal model convergence can still be obtained with careful observation and testing. Our Funk SVD model is now ready to make predictions.

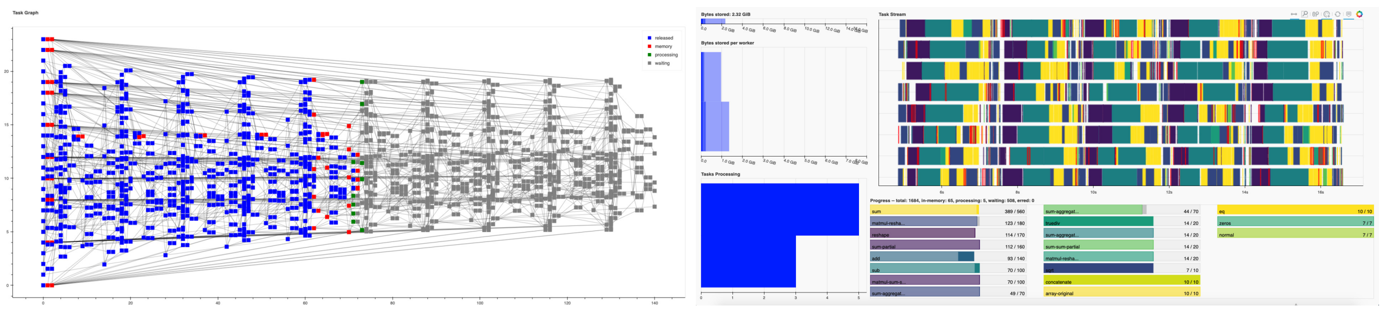

Fig. 3. **Visualization of the Funk SVD model training in parallel running on 10 epochs.** On the left side of the figure we can see a task graph where most of the tasks are executed parallelly, while on the right side we can see specific actions currently executing on workers. We can see that most of these actions are happening in parallel. We can also observe the evenly distributed workload on the workers.

6 Alternating Least Squares

6.1 About ALS

The main difference with ALS algorithm from the Funk SVD is that it alternates between minimizing the two loss functions. First it minimizes the error along the user latent factors and after that along the item latent factors [11].

6.2 Implementation

Similar to Funk SVD algorithm implementation we had to try a few approaches to reach a good working solution that scales. After the implementation of the serial algorithm, our initial thought was to use Dask and parallelize the algorithm naively. But this didn't work, since we were not able to mask our user-item matrix in the same manner as before since latent factor updates are happening separately - row-wise and column-wise. The result of that was that our model learned to predict zeros. To solve this issue we had to chunk our user-item matrix in a totally different way - every row in our matrix needs to be a separate chunk so that we can perform the latent factors update separately. We can see the final form of the chunked matrix in figure (4).

After that, we used Dask block mapping function to perform the user and item factors updates separately in parallel. With this, we achieved parallel row-wise/column-wise computations. This approach worked really well on smaller datasets but it didn't scale at all. When presented with a larger dataset the number of parallel tasks exploded since every row of this giant user-item matrix was scheduled to be computed in parallel at the same time, so the Dask workers couldn't handle the workload. Because of that, we decide to abandon this approach and focus on a more standard and scalable one.

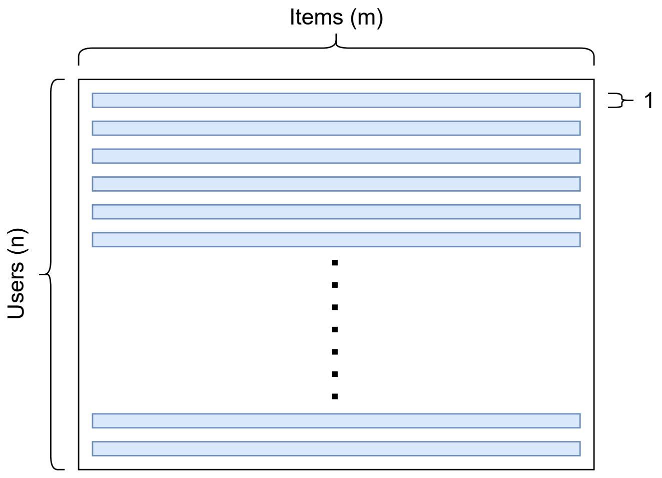

Fig. 4. User-item matrix of dimenison $n \times m$ evenly distributed between in chunks where each chunk is represented by the row of the matrix with size $1 \times m$..

For the second approach, we decided to again use SGD optimization for the value updates because now we know that this approach scales really well. The construction of the user-item matrix and initialization of the biases and latent factors stays the same as in the Funk SVD approach. The only thing that really changes is the updates of the biases and the latent factors during the learning iterations. The difference is that we are now minimizing two loss functions: along with the user latent factors and along with the item latent factors - we are updating them alternately one after the other. This means that we are computing the error two times. The first error computation happens at the start of the iterations and this error is used to update the user factors and biases. After the update, the error is calculated again and this new error is then used to update the item factors and biases. The computations for the updates are the same as in the Funk SVD algorithm. Once again all of the actual Dask computations are done after the last iterations and as we can see in figure (5) this approach scales really well. We have finished with the implementation of our second algorithm so we can move on to the next chapter where we will be evaluating them.

7 Results

In this chapter, we will go over the results of both implemented models on different datasets running on the local and distributed cluster (AWS). Models will be evaluated in terms of prediction accuracy and scalability.

7.1 Prediction accuracy

Accuracy of prediction is measured by the error between the model's predictions and the actual existing values. The lower the value of the error the better is our model's performance. For the error calculations, we used three different measures: Mean Squared Error (MSE), Root Squared Error (RMSE) and Mean Absolute Error (MAE). MSE is denoted as:

$$MSE = \frac{1}{n}\sum_{u,i}(p_{u,i} - r_{u,i})^2 \qquad (1)$$

where $p_{u,i}$ is the expected evaluation for user u in item i and $r_{u,i}$ is the actual evaluation of the user. A total number of evaluations is represented with n. RMSE formula is very similar with added square root over the whole calculation:

$$RMSE = \sqrt{\frac{1}{n}\sum_{u,i}(p_{u,i} - r_{u,i})^2} \qquad (2)$$

And our final evaluation metric MAE is denoted as:

$$MAE = \frac{1}{n}\sum_{u,i}|p_{u,i} - r_{u,i}| \qquad (3)$$

This metrics were used for the optimization of our models and for the final evalvation on our testing set.

7.1.1 *Local cluster*

For the local testing, we fitted both of our models with 100 iterations and 30 latent factors. We choose this number of latent factors because if we increased it our model became more prone to overfitting - we tested this with validation sets. We selected the size of a chunk of 4000 so data is evenly chunked based on the number of users and items. For the learning rate and regularization, we chose a value of 0.001. In figure (6) we can see that our Funk SVD model starts to converge, but we can see that error is still decreasing by some factor. The same goes for the ALS model as we can see in the figure (7), but we can also notice that the ALS model converges way more nicely than the Funk SVD model.

We can see our final results in the table (7.1.1) and we can see that the ALS model has a better evaluation on all three metrics testing on the same testing set. The difference is very small but we can still conclude that in this case, the ALS model is better judging by its prediction accuracy. We can say that both of our models are really good since they clearly outperform the Spark MLlib ALS implementation. But we must consider that we didn't really fine-tune the MLlib model since it was only used as a benchmark of the performance of our implemented models.

Model	MAE	MSE	RMSE
Funk SVD	0.6598	0.8477	0.9207
ALS	0.6376	0.8372	0.9150
MLlib ALS	1.3188	2.5335	1.5917

TABLE 1
Comparison of different model evaulated on the prediction accuracy by three different metrics: MSE, RMSE and MAE. As we can see the model that came on top is ALS by just a slight difference.

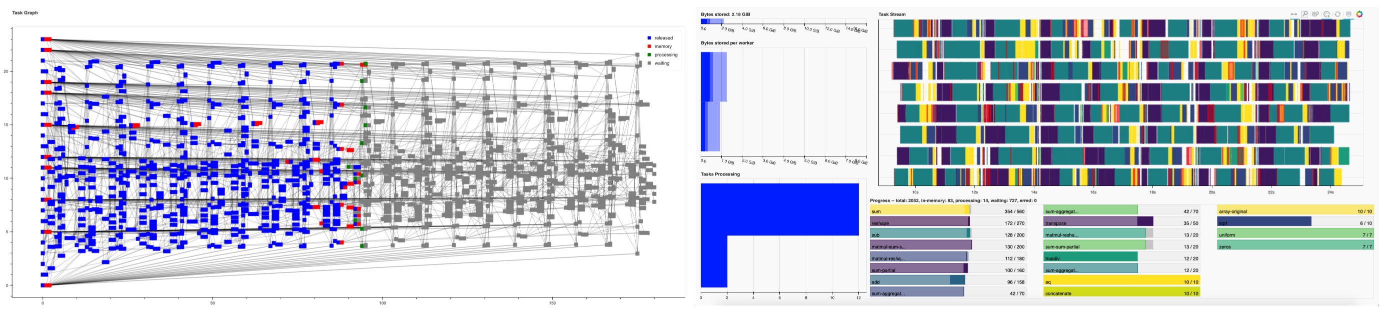

Fig. 5. **Visualization of the ALS model training in parallel running on 10 epochs.** On the left side of the figure we can see a task graph where most of the tasks are executed parallelly, while on the right side we can see specific actions currently executing on workers. We can see that most of these actions are happening in parallel. We can also observe the evenly distributed workload on the workers.

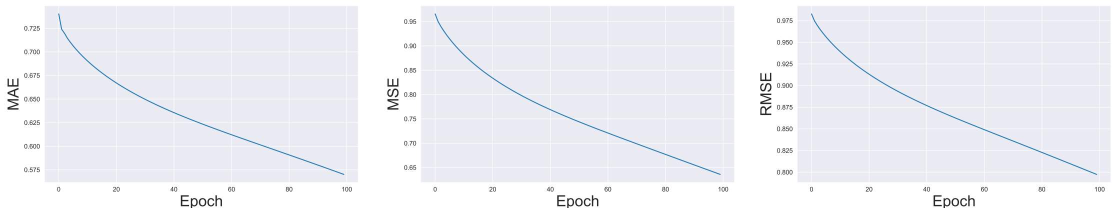

Fig. 6. **Visualization of the training error of the Funk SVD model.** On the figure we can see how the training error decreases during the training process of 100 epochs. The first graph represents MAE, the second one MSE, and the third one RMSE. We can see that all of the errors are decreasing steadily coming close to convergence at the end.

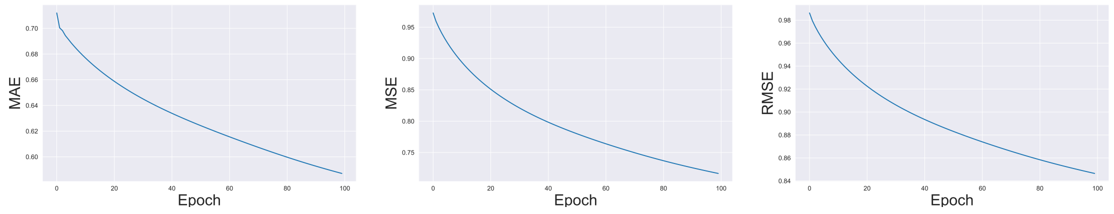

Fig. 7. **Visualization of the training error of the ALS model.** On the figure we can see how the training error decreases during the training process of 100 epochs. The first graph represents MAE, the second one MSE, and the third one RMSE. We can see that all of the errors are decreasing steadily coming close to convergence at the end. We can see that the model converges starts converging more nicely than the Funk SVD model.

7.1.2 Distributed cluster

With the bigger dataset, we had to correct our chunk size because otherwise tasks for our scheduler would explode in numbers and we could get some unwanted behaviour. We choose the chunk size of 5000 and as for the other parameters, they stayed the same. Funk SVD got similar results in terms of prediction accuracy on a bigger dataset running on an AWS distributed cluster with just some minor differences. This is really good since we are really happy with our results and it is a good thing that they scale with the size of the dataset. But we had some problems while running the ALS model. Even with 8 workers each equipped with 11 GB of RAM and 4 CPU cores, the cluster couldn't handle running the model with the bigger dataset. For some reason, workers run out of RAM at the beginning of the computation.

7.2 Scalability

Scalability will be measured and evaluated on metrics of execution time for each iteration of training and the whole fitting process of the model. We measured these values on multiple runs with a different number of workers to see which option performs the best. We tested for scalability with the same model parameters as with accuracy testing. In terms of scalability in using a different number of latent factors, our models are totally scalable since it only changes the computation time for just a small fraction.

7.2.1 Local cluster

From the table 7.2.1 we can see that the fastest run of the Funk SVD training, in this case, was with only one worker. The reason behind that is that communication between workers is very expensive (in general parallelization) because we have to transfer large amounts of data between them. The dataset on which we tested our performance locally was too small to really show the advantages of multiple workers as we can see that one worker worked faster than four workers. This was not expected but since we are really limited with resources using the local machine we couldn't really show the value of multiple workers locally.

# Workers	**1**	**2**	**3**	**4**
Time per epoch(s)	9.47	9.47	9.71	11.04
Execution time(s)	966.18	969.10	989.69	1143.43

TABLE 2
Funk SVD model training times for different number of workers running on local cluster.

In table 7.2.1 we can see a similar situation for the ALS model but in this case, the best run was with two workers. We can also see that ALS takes much more (one-third more) time than the Funk SVD algorithm. Here we can debate which model should be used since ALS performed better in terms of prediction accuracy but for achieving that we have to sacrifice the computation speed. In our opinion the best option would still be to use the Funk SVD model as the difference in prediction accuracy between models is really small. But this depends on the use case and the domain - which of the two metrics is more important: computation speed or prediction accuracy.

# Workers	**1**	**2**	**3**	**4**
Time per epoch(s)	14.39	14.03	16.62	16.40
Execution time(s)	1464.70	1424.08	1693.39	1662.61

TABLE 3
ALS model training times for different number of workers running on local cluster.

7.2.2 Distributed cluster

As our ALS model didn't manage to run on the distributed cluster as mentioned before, we can't really comment on its scalability. But for the Funk SVD, we got some really nice results. In table 7.2.2 we can see the results of training with **4**, **6**, and **8** workers. We can see that our model scales really well in terms of computation speed since with the increase of the number of workers the computation time decreases. If we compare the computation speed between runs with four and six workers we can see that the computation time decreases by almost 13 seconds per iteration which is really good. Even more, the reason that our lower bound number of workers is four is that the training was so slow on a smaller amount of workers. We left our program running on only one worker but even after a few hours we still didn't obtain any results. As for the higher bound of workers, we choose eight since this was the maximum amount of worker instances that we could set up on AWS. We can also see that the time consumed by the communication and data transfer is still a big problem since a run with six workers clearly outperformed the run with eight workers by almost 10 seconds per iteration. We can conclude that our model scales really well with the optimal number of six workers.

# Workers	**4**	**6**	**8**
Time per epoch(s)	33.21	20.09	29.82
Execution time(s)	3344.25	2032.22	3006.53

TABLE 4
Funk SVD model training times for different number of workers running on a distributed cluster on AWS.

8 Conclusion

In conclusion, we can say that our models perform really well based on evaluation with the chosen metrics and in comparison to the Spark MLlib implementation of the algorithms. The models turned out to be really scalable also since the Funk SVD performed really well on a distributed cluster on a large dataset. With the right approach to parallelization, we can achieve high computational speed increases and the option to divide work between different workers in a distributed cluster - this allows us to perform larger tasks that couldn't be done otherwise on a single machine. But we have to be careful when wandering in this waters of parallelization since the learning curve and the lack of documentation can make even the bravest data scientist reevaluate their career choices since programs just don't want to work as expected.

References

[1] F. Isinkaye, Y. Folajimi, and B. Ojokoh, "Recommendation systems: Principles, methods and evaluation," *Egyptian Informatics Journal*, vol. 16, no. 3, pp. 261 – 273, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1110866515000341

[2] "Amazon (company)," https://en.wikipedia.org/wiki/Amazon_company, retrieved: 2021-06-06.

[3] B. Smith and G. Linden, "Two decades of recommender systems at amazon. com," *Ieee internet computing*, vol. 21, no. 3, pp. 12–18, 2017.

[4] M. D. Ekstrand, J. T. Riedl, and J. A. Konstan, *Collaborative filtering recommender systems*. Now Publishers Inc, 2011.

[5] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 173–182.

[6] D. Ferreira, S. Silva, A. Abelha, and J. Machado, "Recommendation system using autoencoders," *Applied Sciences*, vol. 10, no. 16, p. 5510, 2020.

[7] S. Funk, "Funk SVD," https://sifter.org/~simon/journal/20061211.html, accessed: 2021-06-07.

[8] "Dask - dataframe," https://docs.dask.org/en/latest/dataframe.html, retrieved: 2021-06-06.

[9] "Dask - array," https://docs.dask.org/en/latest/array.html, retrieved: 2021-06-06.

[10] "Sparse matrix," https://en.wikipedia.org/wiki/Sparse_matrix#Coordinate_list_(COO), retrieved: 2021-06-06.

[11] K. Liao, "Prototyping a Recommender System Step by Step Part 2: Alternating Least Square (ALS) Matrix Factorization in Collaborative Filtering," https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1, accessed: 2021-06-06.