Short paper

DISGD: A Distributed Shared-nothing Matrix Factorization for Large Scale Online Recommender Systems

Heidy Hazem*, Ahmed Awad**, Ahmed Hassan*, Sherif Sakr**
*Nile University, Egypt
**University of Tartu, Estonia
h.hazem@nu.edu.egy,ahmed.awad@ut.ee,ahassan@nu.edu.egy,sherif.sakr@ut.ee

ABSTRACT

With the web-scale data volumes and high velocity of generation rates, it has become crucial that the training process for recommender systems be a *continuous* process which is performed on live data, i.e., on *data streams*. In practice, such systems have to address three main requirements including the ability to adapt their trained model with each incoming data element, the ability to handle concept drifts and the ability to scale with the volume of the data. In principle, matrix factorization is one of the popular approaches to train a recommender model. Stochastic Gradient Descent (SGD) has been a successful optimization approach for matrix factorization. Several approaches have been proposed that handle the first and second requirements. For the third requirement, in the realm of data streams, distributed approaches depend on a shared memory architecture. This requires obtaining locks before performing updates.

In general, the success of main-stream big data processing systems is supported by their shared-nothing architecture. In this paper, we propose DISGD, a distributed shared-nothing variant of an incremental SGD. The proposal is motivated by an observation that with large volumes of data, the overwrite of updates, lock-free updates, does not affect the result with sparse user-item matrices. Compared to the baseline incremental approach, our evaluation on several datasets shows not only improvement in processing time but also improved recall by 55%.

1 INTRODUCTION

We are living in the era of data abundance whereby good decisions are backed by data-driven approaches. In addition to business-related decisions, we can use data for our personal daily lives. For example, what products to buy, where to have lunch, and best places to spend our vacations are all decisions that we need to make.

Recommender systems [11] have emerged to predict and suggest objects that could be of interest to the user. In general, recommender systems receive input in the form of user-item rating. These ratings are used to update a rating matrix R where the rows represent the users and columns represent items, where usually R is sparse. Collaborative filtering (CF) [5] is a successful technique to guess user preferences based on R. Matrix factorization-based (MF) CF algorithms have shown to be successful. For example, it was able to win the Netflix prize [2]. MF works by decomposing R into two low-dimension vectors of latent factors. Stochastic Gradient Descent (SGD) is used to optimize the weights of these latent factors. In general, SGD is an iterative algorithm that works on a static data set. As data velocity have accelerated, there has become a crucial need to get recommendations with low latency. Therefore, the need to analyze these data and generate new suggestions moved from an offline task on a finite set of data into an online task on a possibly infinite *stream* of data. Thus, a scalable online recommender system has to address three main requirements [3]: 1) The model must be able to produce a result and be updated after each record has been received without passing over all the past data (*latency*). 2)*Concept drifts* [10] ought to be taken care of by adjusting the model with each instance. 3) Online learning from big data must be processed in a distributed streaming environment (*scalability*).

Vinagre et al. [12] have proposed ISGD as an incremental SGD that needs to process each data element once in a streaming fashion. ISGD addresses the first and second requirements above. Yet, it remains a centralized (one worker) solution. Several approaches have introduced parallel (distributed) variants of (I)SGD [1, 4, 6, 7, 13]. However, the common limitation in these approaches is the need to access a shared memory to update the weights among parallel workers. In an online-setting, the overhead to obtain a lock leads to higher latency. An interesting observation by Recht et al. [9] is that with large data, having a lock-free update mechanism, i.e. lost updates, does not affect the overall performance and SGD finally converges. The authors also prove it. Based on this observation, in this paper, we present DISGD as a distributed shared-nothing variant of ISGD. By utilizing the shared-nothing architecture, we allow the best *scalability* as each worker is independent. In particular, the main contributions of this paper can be summarized as follows:

- DISGD: A distributed shared-nothing incremental stochastic gradient descent for a distributed online recommender system (Section 2),
- A comparative evaluation with the baseline ISGD on several data sets showing the superiority of our approach not just in the processing speed but also in the improved recall (Section 3).

2 DISGD

ISGD [12] is an incremental matrix factorization algorithm that is based on SGD. ISGD works centrally where training data are streamed element-by-element. For every received element, ISGD updates the model. So this algorithm overcomes the first two challenges we mentioned in Section 1. In this section, we describe our approach towards addressing the third challenge, scalability.

2.1 Background

In order to reach a scalable ISGD, we depend on the observation that usually the ratio of items to users is petite. For instance, Netflix data set has millions of users and only thousands of items. We start from the observation that the rating matrix $R^{n \times m}$ is sparse. So, we decompose the rating matrix into two matrices,

© 2020 Copyright held by the owner/author(s). Published in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), March 30-April 2, 2020, ISBN 978-3-89318-083-7 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

 10.5441/002/edbt.2020.32

the users' matrix $U^{n \times k}$ and the items' matrix $I^{m \times k}$ with low-dimension k, where $k << n$ and $k << m$, latent features that underlie the items' rating by users. So, we can predict the rating of user u to item i in $R^{n \times m}$, by calculating the dot product between their vectors as in Formula 1.

$$\hat{r_{ui}} = U_n . I_m^T = \sum_{k=1}^{k} u_{nk} . i_{mk} \tag{1}$$

The two matrices U and I are initialized with Gaussian random values. Then, iteratively, SGD calculates how different their product is from the rating matrix R and then makes an effort to minimize this difference. ISGD is dealing with positive feedback only so the error can be calculated by $err_{ui} = 1 - \hat{R_{ui}}$ and we are following that in our algorithm. Furthermore, the gradient descent algorithm iterates many times and updates the vectors which are the rows of the matrices U and I with the purpose of finding a local minimum of the difference following the loss function formulated in 2 where λ is the regularization parameter.

$$min_{U.,I.} \sum_{(u,i) \in D} (R_{ui} - U_u . I_i^T)^2 + \lambda(\|U_u\|^2 + \|I_i\|^2) \tag{2}$$

To parallelize ISGD by distributing the workload among n_c processors, the rating matrix R has to be divided into several blocks and the blocks get assigned to different processors. The issue here is that two processors working on different blocks may need to update the same column of U and/or I. The blocks must be distributed in a way that avoids conflicting updates. Our proposal to solve this problem, and thus addressing the scalability challenge, is by utilizing a splitting and replication mechanism of users and items vectors.

2.2 Splitting and Replication Mechanism

Receiving the rating interactions from users formulated as $<u, i, r>$, Algorithm 1 will distribute the received streamed data of tuples by hashing each record where the user vector and item vector reside over the nodes. For each received tuple $< u, i, r>$, ISGD updates the user vector and the item vector according to the two equations below, where η is the gradient step size.

$$U_u = U_u + \eta(err_{ui} . I_i - \lambda U_u) \tag{3}$$

$$I_i = I_i + \eta(err_{ui} . U_u - \lambda I_i) \tag{4}$$

The aim behind our splitting and replication mechanism is to guarantee that the vectors of users and items are divided over the nodes as it would grow larger than the capacity of one node (central solution). It is assumed that the items are known beforehand. Hence, starting by the item matrix, it is divided into n_i splits (partitions) and each split is replicated over n_c/n_i of the nodes -where n_c is number of nodes in the cluster- while each user vector should exist in n_i of the nodes to always guarantee that a tuple $<u, i, r>$ hits one node where its user and item vectors reside. As a requirement, the number of nodes in the cluster n_c should be equal to $n_i^2 + w.n_i$ where $w \in \mathbb{N}_0$. The distribution technique in Algorithm 1 offloads the storage of vectors to around n_i/n_c of the nodes. For example, when $n_i = 2$, the item matrix I is divided into two halves, each half is stored on half of the nodes. The user matrix U is divided over $n_c/2$ of the nodes and each user vector should exist in two nodes, given $n_i = 2$, over the cluster. Hence, any received tuple is always distributed, in such a way that its user and item vectors are always represented in only one node. Thus, the entire rating matrix will not be needed at any point of time for any single processing task.

Algorithm 1: Parallel ISGD algorithm

Data: data stream of $\{< u, i, r >\} \in D$
Input: N, λ, η, n_i, k
Output: Top N recommended list.

1: **function** DISTRIBUTINGFN(u,i,n_i)
2: $n_c = n_i^2 + w.n_i$
3: $iList \leftarrow$ Map hashing of item id to its $n_c/n_i of nodes$
4: $uList \leftarrow$ Map hashing of user id to its $n_i nodes$
5: key ← Get the common node from uList and iList
6: $outStream < key, u, i, rate >$
7: **end function**

while *receiving {< u,i,r >} ∈ D on each node* **do**
DISTRIBUTINGFN(u,i,n_i)
if $u \notin$ *Rows(U)* **then**
$U_u \leftarrow$ Vector(size : k);
$U_u \sim \mathcal{N}(0,0.1)$;
end
if $i \notin$ *Rows(I)* **then**
$I_i \leftarrow$ Vector(size : k));
$I_i \sim \mathcal{N}(0,0.1)$;
end
$err_{ui} \leftarrow 1 - U_u . I_i^T$;
$U_u \leftarrow U_u + \eta(err_{ui} . I_i - \lambda U_u)$;
$I_i \leftarrow I_i + \eta(err_{ui} . U_u - \lambda I_i)$
end

Algorithm 1 describes how we scale ISGD by means of splitting and replication of the users and items vector. A new top N recommendations list is generated every time a tuple is received. Based on the distributing mechanism shown in Figure 1. This function accords a key to the tuple for maintaining that the pair of user and item vector exists in one node while the single item vector should be in n_c/n_i nodes and the single-user vector should reside in n_i of nodes. This key is produced by hashing the user and item then mapping the hashing output to a predefined list of n_i nodes and n_c/n_i nodes and get the common node number to be the key whereby this key is used for distributing. This node processes the received data and outputs top N recommendations. This particular node only processes $1/n_i$ of the items matrix I that is received in the hashing process described earlier. This does not mean that this particular user will always get his recommendation from this node based on the same items.

It is a random process, based on which $1/n_i$ of the items stored in which node that tuple hits. Moreover, the repetition of the user vector helps offload the storage, making it possible for the algorithm to recommend items for user from different n_i pools of candidates which boosts our recommendation algorithm by giving a wide view for all the items. Algorithm 1 does not need to synchronize between the n_i same user's vectors or n_c/n_i same item's vectors repeatedly stored in the nodes, as Figure 1 shows. It has been proved by Recht et al. [9] that SGD algorithm running over parallel processors with shared memory can converge when the threads overwrite each other and calculate gradient using the outdated current solution which leads to asynchronous machine learning algorithms. Keeping the vectors asynchronous accomplishes two important things, first, it makes DISGD faster and avoids any synchronization or need for lock management.

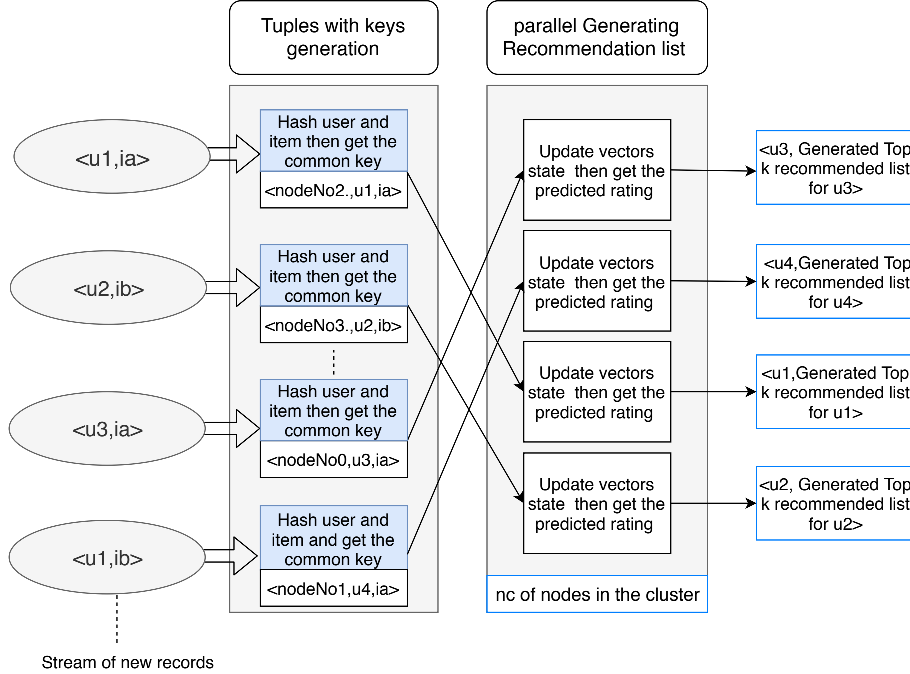


Figure 1: Overview of DISGD collaborative filtering

Algorithm 2: Prequential online evaluator using recall

(1) Recommend top-N recommendation list for the user's coming interaction if the user is known otherwise move to step 3.
(2) score top-N recommendation list based on the coming item i using recall.

$$Recall@N = \begin{cases} 1, \ i \in topNrecommendationlist \\ 0, \ i \notin top - Nrecommendationlist \end{cases}$$

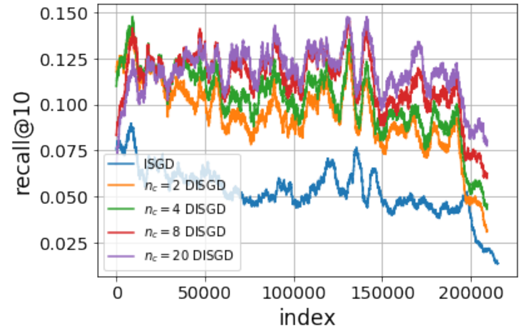
(3) Updated the vectors with the coming instance

---

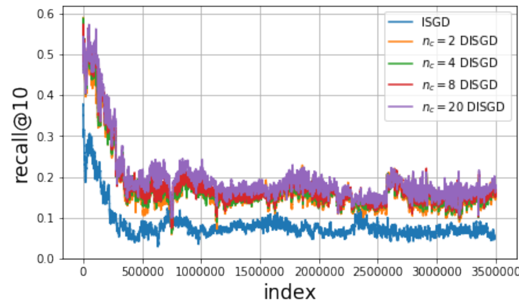
3 EVALUATION

We evaluate DISGD against ISGD as a baseline. The evaluation experiments are done without handling cold start problem as it is not our concern in this paper. We follow prequential evaluation[8] which is suitable and mostly used for streaming algorithms. Prequential evaluation works as follows: for every received instance; it is used first for testing then feed the model with it for training. Specifically, we are following prequential evaluation for streaming recommender systems proposed by Vinagre et al. [12] using the recall evaluation metric which gives indication of how many true positive hits from the user side to the recommendation list by measuring the ratio of relevant items recommended to the total. We compute recall as per Algorithm 2.

The hyperparameter values of equations 3 and 4 used in our experiments are $\lambda = 0.01$, $\mu = 0.05$. We compute the recall with $N = 10$ and set the number of latent features to $k = 10$. DISGD has been implemented on top of Apache Flink version 1.8.1 deployed in a standalone cluster mode with 64 workers. Each worker is a single core running at 2.3 *GHz* with 30 *GB* of main memory. To run the baseline ISGD, we implemented it also as a Flink application and force it to run on a single worker. All the code of our experiments are available for reproducibility [1].



(a) Recall@10 for Moveilens 1M



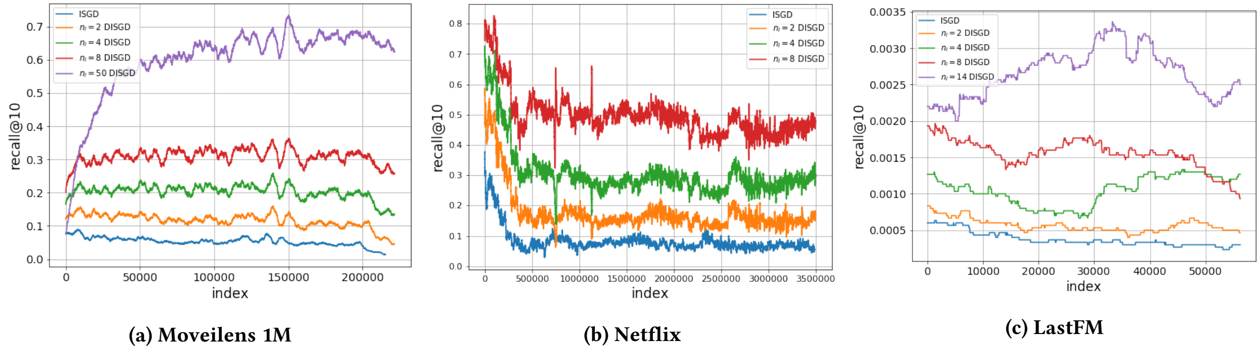
(b) Recall@10 for Netflix

Figure 2: Development of recall@10 testing different n_c. The plotted lines relate to a moving average of the recall@10 got for every recommendation with window size w=5000 with replication factor $n_i = 2$

Data Sets. For our experimntal evaluation, we have used three popular datasets: `Movielens` 1M[2], `Netflix`[3] and `last.FM`[4].

[1] https://github.com/DataSystemsGroupUT/DISGD

[2] https://grouplens.org/datasets/movielens/1m/
[3] https://www.kaggle.com/netflix-inc/netflix-prize-data
[4] https://www.last.fm/



(a) Moveilens 1M

(b) Netflix

(c) LastFM

Figure 3: Development of recall@10 testing different n_i. The plotted lines relate to a moving average of the recall@10 got for every recommendation with window size w=5000 with different replication factor

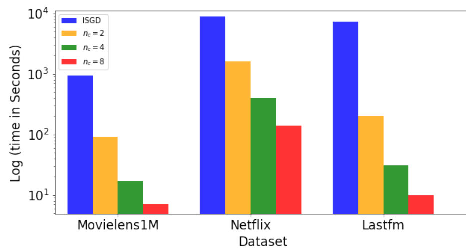


Figure 4: Comparison between the processing time for ISGD and DISGD with different n_i applied on Movielens, Netflix and lastFM datasets.

Movielens and Netflix are notable datasets of rating films and they are sparse. Both the dataset's schema consists of *user*, *item*, *rating* (from 1 to 5) and *time-stamp*. The same preprocessing is done for both datasets. The datasets have been ordered chronologically as indicated by the timestamp for capturing the pattern of how the user interacts with the items consecutively and as our algorithm depends on positive feedback the datasets have been filtered out from any records with a rating under 5. LastFM dataset contains records of listening to music tracks. We extracted the tuples of *<user,trackID>* assuming that the occurrence of the pair as positive feedback and the dataset has been ordered by timestamp.

Experiments and Results. We have run two main experiments. In the first one, we fixed $n_i = 2$; which means that each item vector exists with two versions each on half of the nodes. As per condition in our mechanism, the nodes in the cluster should be n_c=4+2w. We have varied the value of w. The results are reported in Figure 2 showing the results of summing a moving average recall SMA with window size 5000 elements for data sets Movielens 1M and Netflix. We can clearly observe that DISGD achieves significantly a higher recall than the baseline. Obviously, increasing the number of nodes n_c with the same n_i results in higher recall. The recall slightly improves with increasing n_c. The same observation for enhanced recall applies to Netflix.

Regarding the second experiment, DISGD has been tested using different replication factor n_i values with minimum $n_c = n_i^2$. In particular we run our experiments with $n_i \in \{2, 4, 8, 50\}$[5], for `Movielens 1M` dataset and with $n_i \in \{2, 4, 8\}$ for `Netflix`. `LastFM` is tested with $n_i \in \{2, 4, 8, 14\}$[5]. The model has been evaluated using SMA recall at $N = 10$. The results of Figure 3 shows that it is obvious that our approach can scale with different replication factors with enhanced recall in comparison to ISGD.

In general, *processing time* is a major factor in handling streaming data. The x-axis of the graph in Figure 4 shows the three data sets while the log processing time in seconds is on the y-axis. The results show that processing time reduces significantly from ISGD to DISGD and the time decrease dramatically when n_c has risen. It is observed that DISGD is between $6 - 15$ times faster than ISGD with respect to the data sets and the parallelism factor n_c while keeping a significantly higher recall.

4 CONCLUSION AND FUTURE WORK

In this paper, we presented `DISGD`, a distributed shared-nothing variant for stochastic gradient descent for streaming data. Our solution allows much lower latency in serving for recommender systems. However, as with other streaming applications, the data distribution change might lead to skewness in the load on workers. Load rebalancing techniques already exist in literature, however, the effect of moving/merging state on the performance of the algorithm is unknown and is an interesting subject for our future work.

REFERENCES

[1] Muqeet Ali, Christopher C Johnson, and Alex K Tang. 2011. *Parallel collaborative filtering for streaming data.* Technical Report.

[2] Robert M Bell and Yehuda Koren. 2007. Lessons from the Netflix prize challenge. *SiGKDD Explorations Newsletter* 9, 2 (2007), 75–79.

[3] András A Benczúr, Levente Kocsis, and Róbert Pálovics. 2018. Online machine learning in big data streams. *arXiv preprint arXiv:1802.05872* (2018).

[4] Badrish Chandramouli, Justin J Levandoski, Ahmed Eldawy, and Mohamed F Mokbel. 2011. StreamRec: a real-time recommender system. In *SIGMOD.*

[5] David Goldberg et al. 1992. Using collaborative filtering to weave an information tapestry. *Commun. ACM* 35, 12 (1992).

[6] Shohei Hido, Seiya Tokui, and Satoshi Oda. 2013. Jubatus: An open source platform for distributed online machine learning. In *NIPS 2013 Workshop on Big Learning.*

[7] Mu Li et al. 2014. Scaling distributed machine learning with the parameter server. In *OSDI.*

[8] Robert Nishihara et al. 2017. Real-Time Machine Learning: The Missing Pieces. In *HotOS 2017.* ACM, 106–110.

[9] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems.* 693–701.

[10] Paul Resnick and Hal R Varian. 1997. Recommender systems. *Commun. ACM* 40, 3 (1997), 56–59.

[11] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. Introduction to recommender systems handbook. In *Recommender systems handbook.* Springer, 1–35.

[12] João Vinagre, Alípio Mário Jorge, and João Gama. 2014. Fast incremental matrix factorization for recommendation with positive-only feedback. In *International Conference on User Modeling, Adaptation, and Personalization.*

[13] Kais Zaouali et al. 2018. Distributed Collaborative Filtering for Batch and Stream Processing-Based Recommendations. In *OTM.*

[5]The experiment with $n_i = 50$ and $n_i = 14$ have been applied separately on a larger cluster