Distributed Collaborative Hashing and Its Applications in Ant Financial

Chaochao Chen
Ant Financial Services Group
Hangzhou, China, 310099
chaochao.ccc@antfin.com

Ziqi Liu
Ant Financial Services Group
Hangzhou, China, 310099
ziqiliu@antfin.com

Peilin Zhao
Ant Financial Services Group
Hangzhou, China, 310099
peilin.zpl@antfin.com

Longfei Li
Ant Financial Services Group
Hangzhou, China, 310099
longyao.llf@antfin.com

Jun Zhou
Ant Financial Services Group
Hangzhou, China, 310099
jun.zhoujun@antfin.com

Xiaolong Li
Ant Financial Services Group
Hangzhou, China, 310099
xl.li@antfin.com

ABSTRACT

Collaborative filtering, especially latent factor model, has been popularly used in personalized recommendation. Latent factor model aims to learn user and item latent factors from user-item historic behaviors. To apply it into real big data scenarios, efficiency becomes the first concern, including offline model training efficiency and online recommendation efficiency. In this paper, we propose a **D**istributed **C**ollaborative **H**ashing (**DCH**) model which can significantly improve both efficiencies. Specifically, we first propose a distributed learning framework, following the state-of-the-art *parameter server* paradigm, to learn the offline collaborative model. Our model can be learnt efficiently by distributedly computing subgradients in minibatches on workers and updating model parameters on servers asynchronously. We then adopt hashing technique to speedup the online recommendation procedure. Recommendation can be quickly made through exploiting lookup hash tables. We conduct thorough experiments on two real large-scale datasets. The experimental results demonstrate that, comparing with the classic and state-of-the-art (distributed) latent factor models, DCH has comparable performance in terms of recommendation accuracy but has both fast convergence speed in offline model training procedure and realtime efficiency in online recommendation procedure. Furthermore, the encouraging performance of DCH is also shown for several real-world applications in Ant Financial.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; **Retrieval efficiency**;

KEYWORDS

Collaborative; latent factor model, matrix factorization; hashing; parameter server

KDD 2018, August 19–23, 2018, London, United Kingdom

ACM ISBN 978-1-4503-5552-0/18/08.
https://doi.org/10.1145/3219819.3219844

ACM Reference Format:
Chaochao Chen, Ziqi Liu, Peilin Zhao, Longfei Li, Jun Zhou, and Xiaolong Li. 2018. Distributed Collaborative Hashing and Its Applications in Ant Financial. In *KDD 2018: 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219819.3219844

1 INTRODUCTION

With the rapid growth of E-commerce and other online applications, all kinds of information appear on the internet. Meanwhile, more and more users search, buy, and even study through online platforms. Naturally, it becomes much more difficult for users to find their desired items (information), which is known as the information overload problem. Personalized recommendation is widely exploited in almost all the internet companies due to its great ability to solve the information overload problem [6, 28].

Collaborative Filtering (CF) is one of the most popular techniques in personalized recommendation, and its main assumption is that users who behave similarly on some items will also behave similarly on other items [26, 32]. Among CF, model-based methods, especially latent factor models [1, 11, 25], draw a lot of attention, which mainly consist of two steps, i.e., offline model training procedure and online recommendation procedure. The **offline model training procedure** focuses on learning user and item latent factors from the known user-item actions, e.g., buy, rate, and click. Note that user and item latent factors are usually real-valued vectors with the same dimension and preserve the preferences of user and item respectively [12, 20]. The **online recommendation procedure** focuses on ranking the top k items which have the most similar preferences with the target user. Take Figure 1 for example, during the offline model learning procedure, we learn that *Bob* and *Chris* have similar latent factors, since they both like scientific movies. Similarly, we get that *Matrix* and *Inception* have similar latent vectors, since they both belong to scientific movies. Here, we assume user and item latent factors are 3-dimensional binary vectors for simplification. During the online recommendation procedure, we will naturally recommend *Inception* to *Bob* since their latent vectors are the same.

To apply collaborative models in practical big data scenarios, efficiency becomes the first challenge, which includes both offline model training efficiency and online recommendation efficiency.

	Titantic	Matrix	Inception	Serendipity
Alice	1			1
Bob		1		
Chris		1	1	

(a) Original data

user vector	Alice	1	0	0
	Bob	0	1	1
	Chris	0	1	1
item vector	Titantic	1	0	0
	Matrix	0	1	1
	Inception	0	1	1
	Serendipity	1	0	0

(b) Offline model training

Bob (011)	Titantic (100)	N
	Matrix (011)	watched
	Inception (011)	Y
	Serendipity (100)	N

(c) Online recommendation

Figure 1: Toy exmaple. Using latent factor models to make recommendation includes two parts, i.e., offline model training and online recomendation.

On one hand, although most existing latent factor models scale linearly with the size of training dataset, a single machine has very limited memory and CPU. Thus, it is still quite time-consuming to frequently update a collaborative model on a single machine due to the large user and item numbers in real applications. On the other hand, online recommendation needs to calculate the similarities of latent factors between the target user and all the candidate items. This ranking procedure always needs to be done in realtime, and thus becomes a serious problem when the size of candidate items is very large.

To solve the efficiency challenge of the existing collaborative models, in this paper, we propose a Distributed Collaborative Hashing (DCH) model. For offline model training, we develop a distributed framework following the state-of-the-art parameter server paradigm [14, 15, 29]. Specifically, our model can be learnt efficiently by distributedly computing subgradients in minibatches on workers and updating model parameters on servers asynchronously. After that, our model outputs the hash codes as latent factors for each user and each item, which makes them easy to store, comparing with the real-valued latent vectors. For online recommendation, hashing techniques [13, 35] are able to make efficient online recommendations with user and item hash codes. This ranking procedure usually has linear and even constant time complexity by exploiting lookup hash tables [33] and thus satisfies the realtime requirement of online recommendation. We finally conduct thorough experiments on two large-scale datasets, i.e., the public *Netflix* dataset and the real user-merchant payment data in Ant Financial.

Our main contributions are summarized as follows: (1) We propose an efficient model named Distributed Collaborative Hashing (DCH), which offers the ability of efficient offline model training and online recommendation. To the best of our knowledge, this is the first attempt in literature to address the efficiency problems of both offline model training and online recommendation simultaneously. (2) We develop a distributed learning algorithm, i.e., asynchronous stochastic gradient descent with bounded staleness, to optimize DCH using the state-of-the-art parameter server paradigm. (3) The experimental results on two large-scale real datasets demonstrate that, comparing with the classic and state-of-the-art (distributed) latent factor models, our model has both (a) comparable accuracy performance, and (b) fast convergence speed in offline model training procedure and realtime efficiency in online recommendation procedure. (4) We have successfully deployed DCH into several real-world applications in Ant Financial and achieved encouraging performance.

2 BACKGROUND

In this section, we review background knowledge in three groups, i.e., (1) latent factor model, (2) hashing technique, and (3) parameter server.

2.1 Latent Factor Model

Latent factor model aims to learn user and item latent factors from the existing user-item action histories and other additional information, e.g., user social relationship, item content information, and contextual information [1, 4, 5, 8, 11, 19, 20, 25]. Take a classic latent factor model, i.e., matrix factorization [20], for example, it learns user and item latent factors through regression on the existing user-item ratings, which can be formalized as,

$$\arg\min_{u_i, v_j} \sum_{i,j} \left(r_{ij} - u_i^T v_j\right)^2 + \lambda \left(\sum_i ||u_i||^2 + \sum_j ||v_j||^2 \right), \quad (1)$$

where u_i and v_j denote the latent factors of user i and item j, respectively, and r_{ij} denotes the known rating of user i on item j. We will describe other parameters in details later.

Most existing latent factor models aim to learn the real-valued user and item latent factors. Although they have good performance in terms of recommendation accuracy and scale well during offline training, it is difficult for them to rank the top k neighbors efficiently during online recommendation. On one hand, it needs lots of space to store the real-valued user and item latent factors. On the other hand, it is time-consuming to calculate user-item predictions and further rank scores to make recommendations. Hashing technique provides an efficient way to solve this problem.

2.2 Hashing Technique

Hashing technique is aimed at learning binary hash codes of data entities and has been proven a promising approach to solve the nearest neighbor search problem [13, 17, 21, 27, 35]. It not only makes efficient in-memory storage of massive data feasible, but also makes the time complexity of the nearest neighbor search problem linear and even constant by exploiting lookup hash tables

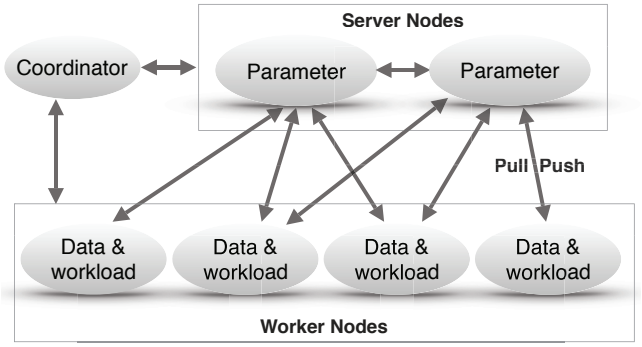


Figure 2: Framework of parameter server, including machines layout and their communications.

[33]. Recently, there is a trend to adopt hashing techniques in personalized recommendation scenarios for better recommendation efficiency [10, 16, 22, 31, 36–38]. They are mainly divided into two categories: (1) first relax the solution from binary space $\{-1, 1\}$ to real space $[-1, 1]$, and then learn real-valued user and item latent factors by using the same way as the existing latent factor models, and finally round or rotate them back to binary hash codes [37, 38]; (2) directly learn user and item binary hash codes by alternatively optimizing subproblems [16, 36].

Although existing hashing based approaches efficiently solve the online recommendation problem, it is difficult for them to scale to large datasets in practice, since they do not support distributed model learning. Parameter server appears to solve this problem.

2.3 Parameter Server

Parameter server is the state-of-the-art distributed learning framework [14, 15, 29]. It contains a coordinator and two other groups of computers, i.e., servers and workers, as is shown in Figure 2. The coordinator controls the start and terminal of the algorithm based on a certain condition, e.g., the number of iterations. The servers are in charge of storing and updating model parameters of an algorithm, e.g., user and item latent factors of matrix factorization. The workers sequentially (or randomly) load and process training data and calculate the changes of the model parameters, e.g., the gradients of user and item latent factors when using gradient descent to optimize matrix factorization. Meanwhile, communications between workers and servers make sure the model parameters are correctly updated. Specifically, communication is performed using the following two operations, (1) **Pull(key, value)**. Before loading data, workers first pull the up-to-date model parameters from servers, and individually update these parameters. Note that, before pull operation, workers first check which parameters will be updated in this minibatch, and then only pull these parameters from servers. This sparse communication strategy greatly reduces the communication time, especially when model parameters are extremely large. (2) **Push(key, value)**. After workers load data and calculate the changes of parameters, they push the changes of parameters to servers, and then let servers update these parameters.

The coordinator allows workers and servers to asynchronously update model parameters with bounded staleness, and thus is able to make fully use of the memory and CPU of each machine.

3 DISTRIBUTED COLLABORATIVE HASHING

In this section, we first formulate our problem. We then describe the collaborative hashing model that aims to learn user and item hash codes. Next, we propose the distributed optimization method to learn the collaborative hashing model. Then, we present our distributed implementation in details. Finally, we describe online recommendation and end this section with a discussion.

3.1 Problem Formulation

Collaborative Filtering (CF) solves the information overload problem through recommending latent interesting items to target users. To do this, CF first learns the real-valued latent factors of users and items from the past user-item action histories, and then makes recommendation through matching users' and items' latent factors. In contrast, collaborative hashing aims to learn the binary-valued latent factors of users and items. Formally, let $\mathcal{U}$ and $\mathcal{V}$ be the user and item set with M and N denoting user size and item size, respectively. Let r_{ij} be the known rating of user $i \in \mathcal{U}$ on item $j \in \mathcal{V}$. Usually, r_{ij} is a real value in a certain region, and without loss of generality, we assume r_{ij} ranges in $[0, 1]$ in this paper. Let $\mathcal{O}$ be the training dataset, where all the user-item ratings in it are known. We also use $\mathcal{O}_u$ and $\mathcal{O}_v$ to denote the users and items in the training dataset, respectively. We further let $\mathbf{U} \in \{-1, 1\}^{K \times M}$ and $\mathbf{V} \in \{-1, 1\}^{K \times N}$ be the user and item latent feature matrices, with their column vectors u_i and v_j be the K-dimensional binary hash codes for user i and item j, respectively. That is, $u_i \in \{-1, 1\}^K$ and $v_j \in \{-1, 1\}^K$, where K is the length of the hash codes.

Collaborative hashing aims to learn hash codes for each user $i \in \mathcal{U}$ and each item $j \in \mathcal{V}$, and then makes recommendation through matching the hash codes of users and items.

3.2 Collaborative Hashing

To make recommendation using the learnt hash codes, user and item hash codes need to preserve the preferences between them, i.e., the similarity between their hash codes should directly denote the similarity of their preferences. For two hash codes, e.g., u_i and v_j, their similarity can be estimated by their Hamming distance, i.e., the common bits in u_i and v_j, that is,

$$sim(u_i, v_j) = \frac{1}{K} \sum_{k=1}^{K} \mathbb{1}\left(u_i^{(k)} = v_j^{(k)}\right) = \frac{1}{2} + \frac{1}{2K} u_i^T v_j, \tag{2}$$

where $\mathbb{1}(\cdot)$ is the indicator function that equals to 1 if the expression in it is true and 0 otherwise. To learn the approximate solutions of u_i and v_j, we introduce the following objective function,

$$\begin{aligned} \underset{u_i, v_j \in \{-1,1\}^K}{\arg\min} \mathcal{L} = & \sum_{i \in \mathcal{O}_u, j \in \mathcal{O}_v} \left(r_{ij} - \frac{1}{2} - \frac{1}{2K} u_i^T v_j\right)^2 \\ & + \lambda \left(|| \sum_{i \in \mathcal{O}_u} u_i ||_F^2 + || \sum_{j \in \mathcal{O}_v} v_j ||_F^2 \right), \end{aligned} \tag{3}$$

where $\lambda \geq 0$ and $|| \cdot ||_F^2$ denotes the Frobenius norm [38].

Equation (3) has an intuitive explanation. Its first term constrains that a user-item rating is proportional to their similarity, i.e., the bigger a user-item rating is, the more similar their hash codes are. The second term requires the binary codes to be balanced, i.e., they have the equal chance to be -1 or 1, and λ controls the balanced degree. The balance constraint is equivalent to maximizing the entropy of each bit of the binary codes, which indicates that each bit carries as much information as possible [38].

Relaxation is widely adopted to find the approximate solutions of binary values [34]. We first relax the solution space from $\{-1,1\}^K$ to $[-1,1]^K$, and then apply the continuous optimization techniques, e.g., stochastic gradient descent or coordinate descent, to solve Eq.(3), and at last, round the learnt real-valued solutions into $\{-1,1\}^K$.

Given the relaxed problem, we take the gradient of $\mathcal{L}$ with respect to u_i and v_j and get,

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial u_i} &= -\frac{1}{K}\sum_{j\in O_v}\Big(r_{ij}-\frac{1}{2}-\frac{1}{2K}u_i^T v_j\Big)v_j + 2\lambda\sum_{i'\in O_u} u_{i'},\\ \frac{\partial \mathcal{L}}{\partial v_j} &= -\frac{1}{K}\sum_{i\in O_u}\Big(r_{ij}-\frac{1}{2}-\frac{1}{2K}u_i^T v_j\Big)u_i + 2\lambda\sum_{j'\in O_v} v_{j'}.\end{aligned} \tag{4}$$

In this paper, we choose Stochastic Gradient Descent (SGD) to optimize our model. Suppose α is the learning rate, u_i and v_j are updated as,

$$\begin{aligned}u_i &\leftarrow u_i - \alpha\frac{\partial \mathcal{L}}{\partial u_i},\\ v_j &\leftarrow v_j - \alpha\frac{\partial \mathcal{L}}{\partial v_j}.\end{aligned} \tag{5}$$

Obtaining Binary Codes. After we solve the approximate solution of **U** and **V**, we have the real-valued user and item latent vectors, i.e., u_i and $v_j \in [-1,1]^K$. Thus, we need to round the real-valued user and item latent factors into binary codes, i.e., $\widetilde{u}_i$ and $\widetilde{v}_j \in \{-1,1\}^K$. Following the work in [38], we round real-valued u_i and v_j to their closest binary vectors $\widetilde{u}_i$ and $\widetilde{v}_j$, so that the learnt binary codes are balanced. That is,

$$\begin{aligned}\widetilde{u}_i^{(k)} &= \begin{cases}1, & {u_i}^{(k)} > \text{median}\,({u_i}^{(k)} : i \in \mathcal{U}),\\ -1, & \text{otherwise},\end{cases}\\ \widetilde{v}_j^{(k)} &= \begin{cases}1, & {v_j}^{(k)} > \text{median}\,({v_j}^{(k)} : j \in \mathcal{V}),\\ -1, & \text{otherwise},\end{cases}\end{aligned} \tag{6}$$

where k denotes the k-th bit in the binary codes and median$(\cdot)$ represents the median of a set of real numbers.

3.3 Distributed Optimization—Asynchronous Stochastic Gradient Descent with Bounded Staleness

To fully make use of the memory and CPU of all the workers and servers, we adopt the idea of asynchronous stochastic gradient descent with bounded staleness to optimize our model. "Asynchronous" means that all the workers independently run SGD on mini-batch datasets. Specifically, each worker iteratively performs **a SGD operation**, i.e., it pulls model parameters (i.e., user and item latent factors) from servers, loads mini-batch training data, calculates gradients, and pushes the gradients (i.e., model changes) to servers. Meanwhile, each server updates the model parameters stored in it once it receives the model gradients from workers. Since there are multi-worker running the SGD procedures asynchronously, each worker will probably not send the most recent model gradients to servers. The idea of "bounded staleness (delay)" has been extensively used in distributed frameworks [14]. It makes a compromise between total synchronization and asynchronization, which means that servers will wait for all the workers to synchronize once after a certain period. We use parameter P to denote the period of synchronization, i.e., synchronize once after each worker has done P SGD procedures.

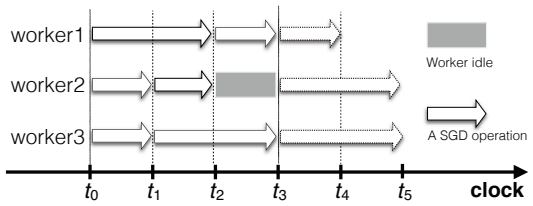


Figure 3: An illustration of asynchronous SGD with bounded staleness ($P = 2$).

Figure 3 shows the illustration of an asynchronous SGD with bounded staleness when $P = 2$, where we omit the time of server operations for conciseness. Three workers, i.e., W_1, W_2, and W_3, start to perform SGD operations at t_0. W_1 takes three time slides (from t_0 to t_3) to finish two SGD iterations—two time slides (from t_0 to t_2) for the first iteration and one time slide (from t_2 to t_3) for the second iteration. Similarly, W_2 takes two time slides (from t_0 to t_2) to finish two SGD iterations, and W_3 takes three time slides (from t_0 to t_3) to finish two SGD iterations. During these two iterations, all the workers perform SGD operations independently. However, servers will wait for all the workers to synchronize after the second iteration at time t_3, since we set $P = 2$ in this example. After then, all the workers start to perform SGD operations independently again at t_3. From it, we can see that synchronous SGD is a special case of asynchronous SGD with bounded staleness when $P = 1$.

Model parameters projection. Model parameters may diverge when performing distributed SGD. To avoid this, gradient-projection approach becomes a choice to limit the set of admissible solutions in the ball of radius $1/\sqrt{\gamma}$ [30]. That is, we project u_i and v_j into this sphere by performing the following update at each synchronization period t, e.g., t_3 in Figure 3, that is

$$\begin{aligned}u_{i,t} &\leftarrow min\left\{1, \frac{1/\sqrt{\gamma}}{||u_{i,t}||}\right\}u_{i,t},\\ v_{j,t} &\leftarrow min\left\{1, \frac{1/\sqrt{\gamma}}{||v_{j,t}||}\right\}v_{j,t}.\end{aligned} \tag{7}$$

where $u_{i,t}$ and $v_{j,t}$ denote latent vectors of u_i and v_j at time t, respectively.

3.4 Implementation

We summarize the implementation sketch of DCH using parameter server in Algorithm 3.1, which contains the following three parts:

Coordinator node controls the status and procedure of the whole algorithm, including starts and terminates of the algorithm, and gives commands to workers and servers. Specifically, it first

Algorithm 1: Implement DCH on parameter server

Input: ratings in training set (O), worker number (W), server number (S), synchronization period (P), minibatch size (B), learning rate (α), projection radius (γ)
Output: user and item hash code matrices **U** and **V**

Coordinator node:
for $s = 1$ *to* S **do**
  Assign server s to do initialization
end
Partition training data O evenly based on W
repeat
  for $p = 1$ *to* P **do**
    for $w = 1$ *to* W, *parallel* **do**
      Assign worker w to do a SGD operation
    end
  end
  Wait all workers to synchronize
  for $s = 1$ *to* S **do**
    Assign server s to do projection operation
  end
until *convergence*;
for $s = 1$ *to* S **do**
  Assign server s to do round operation
end
Worker node w**:**
if *receive a SGD command from the coordinator* **then**
  Load a minibatch size (B) data
  Pull u_i and v_j that appear in this minibatch from server nodes
  Compute the gradients based on Eq.(4)
  Push gradients back to servers
end
Server node s**:**
if *receive initialization command from the coordinator* **then**
  Initialize **U** and **V**
end
if *receive gradients from a worker* **then**
  Update **U** and **V** based on Eq.(5)
end
if *receive projection command from the coordinator* **then**
  Project **U** and **V** based on Eq.(7)
end
if *receive round command from the coordinator* **then**
  Round **U** and **V** based on Eq.(6)
end
return U and **V**

assigns servers to initialize u and v, and then partitions the training data evenly based on W. During the asynchronization period, it assigns workers to do SGD operations independently. At each synchronization point, i.e., when the current iteration number $t\%p = 0$, it waits all workers to synchronize and then assigns servers to do parameter projection. Finally, it assigns servers to do a round procedure after the model converges.

Worker nodes load data and perform computation. Once they receive SGD commands from the coordinator, they first randomly load a minibatch data. They then pull the old user and item factors that will be used in the minibatch data from servers. After that, they calculate the gradients of user and item latent factors, and finally push them to servers.

Server nodes store and update **U** and **V**. They initialize **U** and **V** at the beginning. Then, once receive gradients from a worker, they update the corresponding user and item latent factors. They will also project the solutions of **U** and **V** at each synchronization point. Finally, they round the real-valued solutions to hash codes after the model converges.

3.5 Online Recommendation

After user and item hash codes are learnt, online recommendation is extremely efficient. Hash code length (i.e., dimension of latent factors) and the number of candidate items are two factors that affect online recommendation efficiency. Hamming distance search [24], hashing lookup[27], Hamming ranking [17], and multi-index hashing [21] are widely used for searching nearest neighbors in hashing techniques. Hamming distance search retrieves items with binary codes within a certain Hamming distance to the binary codes of the target user. Hamming distance can be quickly calculated by using the *XOR* operations between user and item hash codes, and thus is constant w.r.t the hash code length. Hashing lookup returns items with binary codes within a certain Hamming distance to the binary codes of the target user from hash tables, and thus its time complexity is constant w.r.t the dataset size. Hamming ranking ranks items according to their hamming distance with the target user and returns fixed items. Speedup can be achieved by fast hamming distance computation, i.e. popcnt function [37]. Multi-index hashing splits the original hashing code into several subcodes and conducts hashing table lookup seperately, which improves hashing lookup. We will further perform experiments in Section 4.4 to show the online recommendation efficiency of hashing techniques comparing with the existing latent factor models.

3.6 Discussion

Recently, graph representation has been drawing a lot of attention in both academic and industry, and can be applied into many tasks such as recommendation [7]. Graph representation aims to learn latent representations of nodes on graphs, which can be seen as an improvement of the classic matrix factorization technique by considering complicated structural information on graph [3]. Existing research has proven that most graph representation methods, e.g., DeepWalk [23] and node2vec [9], are in theory performing implicit matrix factorizations. Intuitively, these methods suffer from online recommendation efficiency problem when applying to recommendation tasks. Therefore, our proposed distributed collaborative hashing can be natually used to solve the efficiency problem. That is, we learn hash code representations instead of real-valued ones for nodes on graph, which is one of our future works.

Table 1: Dataset description

Dataset	#user	#item	#rating
Netflix	480,189	17,770	100,480,507
Alipay	9,111,142	443,043	1,599,288,565

4 EXPERIMENTS AND APPLICATIONS

In this section, we empirically compare the performance of the proposed DCH with the classic and state-of-the-art distributed collaborative models, including recommmendation accuracy performance and recommendation efficiency performance. We also study the parameters on its model performance and describe its the applications at Ant Financial.

4.1 Setting

Datasets. We use two large-scale datasets, i.e., *Netflix* dataset and the real user-merchant payment data in *Alipay*. *Netflix* dataset is famous from the Netflix competition[1] [2], and is widely used due to its publicity and authority. *Alipay* is a product of Ant Financial, which is also the biggest third-party online payment platform in China, and through it users are able to delivery both online and offline payment. Our *Alipay* dataset consists of two parts. The first part is sampled from user-merchant payment records during 2015/11/01 to 2017/10/31 whose ratings are taken as '1'. The second part is sampled from the non-payment data whose ratings are taken as '0'. Combining both parts, we get the *Alipay* dataset. We show their statistics in Table 1.

Metrics. We adopt two metrics to evaluate our model performance, i.e., precision and Discounted Cumulative Gain (DCG). Both of them are extensively used to evaluate the quality of rankings [18, 38]. For each user, precision and DCG are defined as:

$$Precision@k = \frac{Number\ of\ postive\ items\ in\ Top\ k}{k},$$

$$DCG@k = \sum_{i=1}^{k} \frac{2^{r_i} - 1}{log(i+1)},$$

where r_i denotes the rating of the i-th retrieved item. We take items whose ratings are equal to 5 as positive items on *Netflix* dataset, and take each user-merchant payment as a positive data that a user rates a merchant (item) on *Alipay* dataset.

We split both datasets with two strategies: (1) randomly sample 80% as training set and the rest 20% as test set, and (2) randomly sample 90% as training set and the rest 10% as test set. We use *Netflix80* and *Alipay80* to denote the first strategy, and use *Netflix90* and *Alipay90* to denote the second strategy.

Comparison methods. We compare our proposed DCH with the following classic and state-of-the-art distributed models:

- **Matrix Factorization (MF)** is a classic collaborative model [20], and it factorizes the user-item rating matrix into real-valued user and item latent factor matrixes without rounding them into hash codes.
- **MFH** first learns the real-valued user and item latent factors the same way as MF, and then rounds user and item latent factors into hash codes using Eq.(6).
- **Distributed Factorization Machine (DFM)** is an implementation of the state-of-the-art DiFacto model [15] on parameter server.
- **DFMH** first learns the real-valued user and item latent factors the same way as DFM, and then rounds user and item latent factors into hash codes using Eq.(6).

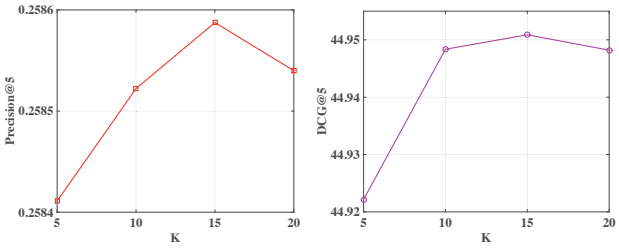
Hyper-parameters. We set worker number $W = 50$, server number $S = 20$, synchronization period $P = 2$, learning rate $\alpha = 0.001$, and minibatch size $B = 1000$. Besides, since our relaxed solution space is $[-1, 1]$, we set projection radius $1/\sqrt{\gamma} = 1$, i.e., $\gamma = 1$. That is, we project the factors of users and items to 1 if they are bigger than 1 and to -1 if they are smaller than -1. For the latent factor dimension K, we find its best value in $\{5, 10, 15, 20, 25, 30\}$. We find the best values of other hyper-parameters in $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}\}$.

4.2 Comparison Results

To verify our model performance, we compare DCH with four classic and state-of-the-art models on both *Netflix* and *Alipay* datasets. During the comparison, we use grid search to find the best parameters of each model. We report the comparison results on *Netflix* in Table 2 and Table 3, where $K = 5$ and $K = 10$ respectively, and report the comparison results on *Alipay* in Table 4 and Table 5, where $K = 5$ and $K = 10$ respectively. From them, we find that:

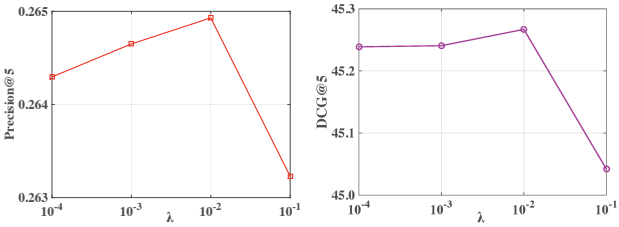
- Recommendation performances of all the models increase with training data size. For example, the Precision@5 of DCH increase 2.52% on *Netflix90* comparing with that on *Netflix80* when $K = 10$. This is due to the data sparsity problem, i.e., recommendation accuracy decays when training data becomes sparser.
- Recommendation performances of all the models are affected by the dimension of latent factor or hash code length. Close observation shows the following results: recommendation accuracy of all the models are better when $K = 10$ than $K = 5$ on *Netflix* datasets. On the contrary, some of them are better when $K = 5$ than $K = 10$ on *Alipay* datasets, especially on *Alipay80*. Note that the rating densities of *Netflix* and *Alipay* are 1.18% and 0.04%, respectively. Different latent factor dimension or hash code length will contains different amount of information. Small dimension/length will cause information loss, while large dimension/length may cause over-fitting. Since the rating density of *Alipay* is much smaller than that of *Netflix*, and *Alipay80* has even sparser ratings, over-fitting is easier to appear on *Alipay80* with the same K. This is why the best value of K on *Alipay* is smaller than that on *Netflix*.
- DCH achieves better results than the other two hash-based methods, i.e., MFH and DFMH, in almost all the cases. This is because, DCH directly optimizes the difference between rating and hamming distance, as is shown in Eq. (2). In contrast, both MF and DFM optimize the difference between rating and the product of user-item's latent factors, as is shown in Eq. (1).
- DCH has the *comparable performance* with the classic and state-of-the-art distributed latent factor models, i.e., MF and DFM. This is consistent with the existing research [38], i.e., hash technique

[1]The dataset is available at: https://www.kaggle.com/netflix-inc/netflix-prize-data/data



(a) Effect on Precision@5 (b) Effect on DCG@5

Figure 4: Effect of K on DCH. Dataset used: *Netflix90*.



(a) Effect on Precision@5 (b) Effect on DCG@5

Figure 5: Effect of λ on DCH. Dataset used: *Netflix80*.

can achieve comparable prediction accuracy performance with the classic latent factor models by using the appropriate values of K.

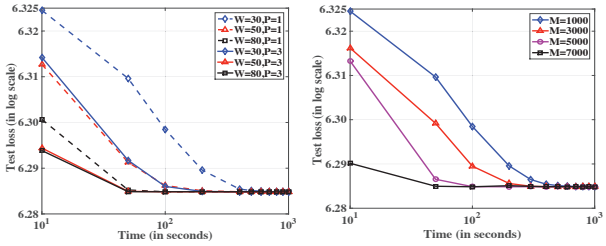
4.3 Parameter Analysis

Our model mainly has five hyperparameters, i.e., hash code length (K), hash code balanced degree (λ), worker number (W), minibatch size (B), and synchronization period (P). We first fix $W = 50$, $B = 1000$, and $P = 2$, and study the effects of K and λ.

Effect of K. In the above sub-section, we have compared each model's performance with different K, we now study its effect on our model performance in details. Figure 4 shows the effect of K on *Netflix90* where $\lambda = 0.01$. We observe that our model performance first increases and then decreases after a certain value of K. This is because the bigger K is, the more information user and item hash codes contain. Thus, our model performance increases with K at first. However, our model tends to be over-fitting when K is too bigger, i.e., $K = 15$. The best value of K can be determined by cross-validation in practice. In general, we find DCH achieves the best performance when K is around 15 on *Netflix*.

Effect of λ. Parameter λ controls the balanced degree of user and item hash codes. The bigger λ is, the more balanced user and item hash codes are. Figure 5 shows the effect of λ on *Netflix80* where $K = 10$. As we can see, our model performance first increases and then decreases after a certain value of λ. This indicates that the accuracy of DCH can be further improved with a good value of λ, since a better user and item hash codes are learnt. The best value of λ can also be determined by cross validation when using DCH.

We then fix $K = 10$ and $\lambda = 0.01$ and study the effects of W, B, and P.



(a) Effect of W (b) Effect of B

Figure 6: Effect of #worker (W) and minibatch size (B) on model convergence speed. Dataset used: *Netflix*.

Effect of W and B. Parameter W controls the worker number, i.e., how many workers perform asynchronous SGD independently. Parameter B controls the minibatch size, i.e., each worker loads and processes B training data during each SGD operation. Both of them control the total size of training data workers can process in a certain epoch. Figure 6 (a) shows the effect of W on model convergence speed, where we fix $B = 1000$, and set $P = 1$ (dash lines) and $P = 3$ (solid lines). Figure 6 (b) shows the effect of B on model converge speed, where we fix $W = 30$ and $P = 1$. From them, we find that more workers and bigger minibatch size can speedup model convergence. The results are reasonable, because our model can process more training data by using more workers and increasing minibatch size, and thus converges faster.

Effect of P. Parameter P controls the period of synchronization, i.e., servers will wait for all the workers to synchronize once after each worker has done P SGD operations. Figure 7 (a) shows the effect of P on model convergence speed, where we fix $B = 1000$, and use $W = 50$ (dash lines) and $W = 80$ (solid lines) respectively. From it, we find that our model converges faster with a bigger P. The speedup ratio is 10 when increasing P from 1 to 5 by fixing other parameters. Workers process SGD procedures independently without waiting for each other during the asynchronization periods, which makes the model converge much faster, especially when P is big. However, we also calculate model variance and find that model becomes relative unstable when P is too big. We show this finding in Figure 7 (b), where we fix $B = 1000$ and $W = 80$. As we explained in Section 3.3, each worker will probably not send the most recent model gradient to servers when using asynchronous SGD with bounded staleness, which may cause the model deviate too much. An appropriate value of P (staleness bound) can not only make model converges faster, but also more stable. This experiment proves the practicalness of asynchronous SGD with bounded staleness.

4.4 Recommendation Efficiency

First, we compare the offline model training efficiency of the classic latent factor model (i.e., MF) and DCH, where we set $B = 50,000$ and $P = 1$. Note that we do not compare DCH with DFM or DFMH here, because DFM, DFMH, and DCH are all distributed algorithms implemented on parameter server and share almost the same training speed. The results are shown in Table 6: DCH significantly decreases the offline training time of MF. Besides, the more workers

Table 2: Performance comparison on *Netflix* datasets ($K = 5$)

Datasets	*Netflix80*				*Netflix90*			
Metrics	Precision@5	Precision@10	DCG@5	DCG@10	Precision@5	Precision@10	DCG@5	DCG@10
MFH	0.2580	0.2473	44.9102	68.2807	0.2644	0.2573	45.2374	69.1247
DFMH	0.2572	0.2472	44.8933	68.2634	0.2641	0.2570	45.2339	69.1099
MF	0.2589	0.2479	44.9351	68.3471	0.2658	0.2584	45.3774	69.3064
DFM	0.2589	0.2481	45.0031	68.4398	0.2658	0.2586	45.3790	69.3621
DCH	0.2584	0.2478	44.9222	68.3279	0.2645	0.2576	45.2383	69.1353

Table 3: Performance comparison on *Netflix* datasets ($K = 10$)

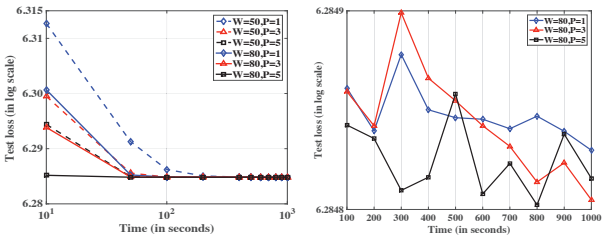
Datasets	*Netflix80*				*Netflix90*			
Metrics	Precision@5	Precision@10	DCG@5	DCG@10	Precision@5	Precision@10	DCG@5	DCG@10
MFH	0.2583	0.2478	44.9184	68.2782	0.2644	0.2572	45.2518	69.1101
DFMH	0.2579	0.2472	44.8933	68.2634	0.2644	0.2571	45.2484	69.1152
MF	0.2587	0.2479	44.9645	68.3807	0.2650	0.2576	45.2911	69.1707
DFM	0.2585	0.2482	44.9404	68.3803	0.2654	0.2576	45.3311	69.2036
DCH	0.2584	0.2479	44.9484	68.3808	0.2649	0.2574	45.2631	69.1472

Table 4: Performance comparison on *Alipay* datasets ($K = 5$)

Datasets	*Alipay80*				*Alipay90*			
Metrics	Precision@5	Precision@10	DCG@5	DCG@10	Precision@5	Precision@10	DCG@5	DCG@10
MFH	0.200322	0.200266	0.590591	0.909867	0.200692	0.200304	0.591730	0.910686
DFMH	0.200443	0.200300	0.591013	0.910227	0.200714	0.200396	0.591818	0.910690
MF	0.200550	0.200319	0.591969	0.911047	0.200992	0.200374	0.593726	0.912198
DFM	0.200565	0.200335	0.592235	0.911388	0.201059	0.200415	0.594371	0.912825
DCH	0.200329	0.200323	0.590738	0.910408	0.200814	0.200403	0.592797	0.911733

Table 5: Performance comparison on *Alipay* datasets ($K = 10$)

Datasets	*Alipay80*				*Alipay90*			
Metrics	Precision@5	Precision@10	DCG@5	DCG@10	Precision@5	Precision@10	DCG@5	DCG@10
MFH	0.200326	0.200274	0.590664	0.909959	0.200809	0.200432	0.592072	0.910968
DFMH	0.200345	0.200289	0.590554	0.909863	0.200671	0.200422	0.591603	0.910554
MF	0.200394	0.200297	0.590823	0.910196	0.200839	0.200435	0.593061	0.911767
DFM	0.200416	0.200328	0.591269	0.910491	0.200921	0.200438	0.593078	0.911937
DCH	0.200356	0.200295	0.590731	0.909967	0.200821	0.200436	0.592246	0.911257



(a) Effect of P on convergence speed (b) Effect of P on model stability

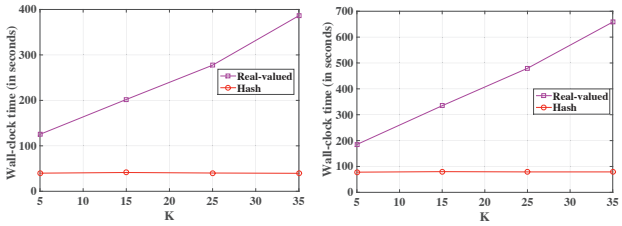
Figure 7: Effect of synchronization period (P) on model convergence. Dataset used: *Netflix*.

DCH uses, the bigger speedup of DCH against MF, since DCH can process more data with more workers and thus converges faster.

Table 6: Comparison of offline training time (in seconds)

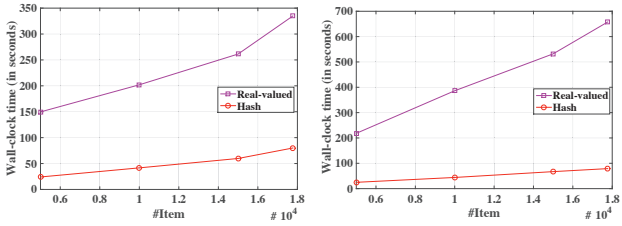
Model	MF	DCH (W=5)	DCH (W=10)
Training time	8,461	2,142	1,351
DCH speedup	-	3.95	6.26

Next, we compare the online recommendation efficiency of the (distributed) latent factor models (i.e., MF and DFM) and DCH. We perform the following experiments on the *Netflix* dataset, that is, we make recommendation from a pool of 17,770 candidate items. For MF and DFM which share the same online speed, we first compute user-item ratings using the learnt real-valued latent factors of user and item, and then sort the ratings, and finally return the top-10 items with highest ratings for each user. We name this approach "real-valued rating rank" and use "real-valued" for abbreviation. The time complexity of computing all the user-item ratings is $O(MNK)$,



(a) Item size: 10000 (b) Item size: 17770

Figure 8: Time cost with different hash code length.



(a) K=15 (b) K=35

Figure 9: Time cost with different item number.

and the time complexity of sorting the top-k items is $O(MN)$. For DCH, we choose Hamming distance search method as described in Section 3.5 and return items within Hamming distance '1' to the binary codes of each user. We name it "Hash" for abbreviation, whose time complexity is $O(MN)$. Note that this experiment is done on a Macbook Air with Intel 64 bit CPU and 8GB memory, and no parallel computing is involved.

Efficiency in terms of hash code length. Figure 8 shows the time cost of real-valued rating rank and Hamming distance search, where hash code length K varies in $\{5, 15, 25, 35\}$ and item size is 17770. As we analyzed in Section 3.5, the time cost of real-valued rating rank scales linearly with the dimension of latent factors. In contrast, the time cost of Hamming distance search is constant w.r.t the hash code length. For example, Hamming distance search only uses 80 seconds to make recommendation for 480,189 users. In other words, it takes only 0.17 milliseconds to make recommendation through 17,770 items for a single user, which totally meets the realtime requirement of online recommendation. These results demonstrate the effectiveness of hashing techniques.

Efficiency in terms of item size. Figure 9 shows the time cost of real-valued rating rank and Hamming distance search, where item size varies in $\{5000, 15000, 20000, 17770\}$ and $K = 15$. From them, we can see that although both real-valued rating rank and Hamming distance search scales linearly with item size, their slopes are different. The time cost of real-valued rating rank increases much faster than Hamming distance search, especially when K is large. This is because the time complexity of real-valued rating rank is $O(MNK)$, while the time complexity of Hamming distance search is $O(MN)$. Moreover, one can also use multi-index hashing or other hashing techniques whose time complexity are constant w.r.t item size when the candidate item size is very large.

In summary, comparing with the traditional MF approaches, DCH is able to significantly improves efficiencies, including both offline model training efficiency and online recommendation efficiency, without losing much accuracy. With such a good property of DCH, it is suitable to be applied into real large-scale scenarios.

4.5 Applications

Our proposed DCH model has been successfully deployed into several real products at Ant Financial, e.g., the prize recommendation of scratch cards during the Chinese New Year 2017 and merchant recommendation in the "Guess You Like" scenario of Alipay. On one hand, the learnt hash codes provide additional useful features for both users and items, and thus can also be integrated into other models, e.g., logistic regression and deep neural network, to further improve their performances. On the other hand, DCH is able to effectively retrieve user's latent interesting items, and thus can be used to match candidate items before ranking or re-ranking. The deployment of DCH can not only improve the click-through rate (CTR) of online recommendations by taking user and item hash codes an additional features, but also significantly decrease the matching time by using the hashing technique.

5 CONCLUSION AND FUTURE WORK

In this paper, we aim to solve the efficiency problem of the existing collaborative models, including offline model training efficiency and online recommendation efficiency. To do this, we proposed a Distributed Collaborative Hashing (DCH) model that aims to learn user and item hash codes from the existing user-item action history. To improve offline model training efficiency, we designed DCH following the state-of-the-art parameter server paradigm. Specifically, we optimized DCH by distributedly computing subgradients on minibatches on workers asynchronously and updating model parameters on servers with bounded staleness. To improve online recommendation efficiency, we adopted hashing technique that has linear or even constant time complexity. Finally, we conducted experiments on two large-scale datasets to prove our model performance and study parameter effects. The results demonstrated that, comparing with the classic and state-of-the-art distributed collaborative models, (1) DCH has comparable performance in terms of recommendation accuracy, and (2) DCH has fast convergence speed in offline model training and realtime efficiency in online recommendation.

Our future work will focus on two directions. One is distributed discrete hash code learning. Our current offline model training approach is two-stage, i.e., first solve the relaxed optimization problem and then rounding off. Inspired by [16, 36], we plan to directly optimize user and item binary hash codes distributedly to further improve model performance. The other is to apply our model into graph representation technique for recommendation. As discussed in Section 3.6, we plan to hash code embeddings for nodes in graphs.

ACKNOWLEDGMENTS

We would like to acknowledge contributions from our colleagues from Alibaba and Ant Financial, including: Xu Chen, Yi Ding, Qing Cui, Jin Yu, and Jian Huang. We would also like to extend our sincere

thanks to the entire Large-Scale-Learning and Distributed-Learning-and-Systems team members. Finally, we thank KDD anonymous reviewers for their helpful suggestions.

REFERENCES

[1] Deepak Agarwal and Bee-Chung Chen. 2009. Regression-based latent factor models. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 19–28.

[2] James Bennett and Stan Lanning. 2007. The netflix prize. In *Proceedings of KDD Cup and Workshop 2007 Aug 12*, Vol. 2007. 35.

[3] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 891–900.

[4] Chaochao Chen, Xiaolin Zheng, Yan Wang, Fuxing Hong, and Zhen Lin. 2014. Context-aware Collaborative Topic Regression with Social Matrix Factorization for Recommender Systems. In *AAAI*. 9–15.

[5] Chaochao Chen, Xiaolin Zheng, Mengying Zhu, and Litao Xiao. 2016. Recommender System with Composite Social Trust Networks. *International Journal of Web Services Research (IJWSR)* 13, 2 (2016), 56–73.

[6] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. 2007. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*. ACM, 271–280.

[7] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time. In *Proceedings of the 27th International Conference on World Wide Web*.

[8] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 69–77.

[9] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 855–864.

[10] Alexandros Karatzoglou, Markus Weimer, and Alex J Smola. 2010. Collaborative filtering on a budget. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 389–396.

[11] Yehuda Koren. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 426–434.

[12] Yehuda Koren, Robert Bell, Chris Volinsky, et al. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[13] Brian Kulis and Kristen Grauman. 2009. Kernelized locality-sensitive hashing for scalable image search. In *Proceedings of the 12th International Conference on Computer Vision*. IEEE, 2130–2137.

[14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *OSDI*. 583–598.

[15] Mu Li, Ziqi Liu, Alexander J Smola, and Yu-Xiang Wang. 2016. DiFacto: Distributed Factorization Machines. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. ACM, 377–386.

[16] Defu Lian, Rui Liu, Yong Ge, Kai Zheng, Xing Xie, and Longbing Cao. 2017. Discrete Content-aware Matrix Factorization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 325–334.

[17] Wei Liu, Jun Wang, Rongrong Ji, Yu-Gang Jiang, and Shih-Fu Chang. 2012. Supervised hashing with kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2074–2081.

[18] Xianglong Liu, Junfeng He, Cheng Deng, and Bo Lang. 2014. Collaborative hashing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2139–2146.

[19] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*. 165–172.

[20] Andriy Mnih and Ruslan Salakhutdinov. 2007. Probabilistic matrix factorization. In *Advances in Neural Information Processing Systems*. 1257–1264.

[21] Mohammad Norouzi, Ali Punjani, and David J Fleet. 2012. Fast search in hamming space with multi-index hashing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 3108–3115.

[22] Mingdong Ou, Peng Cui, Fei Wang, Jun Wang, Wenwu Zhu, and Shiqiang Yang. 2013. Comparing apples to oranges: a scalable solution with heterogeneous hashing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 230–238.

[23] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 701–710.

[24] Justin Ray and Philip Koopman. 2006. Efficient high hamming distance CRCs for embedded networks. In *DSN*. IEEE, 3–12.

[25] Steffen Rendle. 2010. Factorization machines. In *Proceedings of the IEEE 10th International Conference on Data Mining (ICDM)*. 995–1000.

[26] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. 1994. GroupLens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*. 175–186.

[27] Ruslan Salakhutdinov and Geoffrey Hinton. 2009. Semantic hashing. *International Journal of Approximate Reasoning* 50, 7 (2009), 969–978.

[28] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *WWW*. 285–295.

[29] Sebastian Schelter, Venu Satuluri, and Reza Zadeh. 2014. Factorbird-a parameter server approach to distributed matrix factorization. *arXiv preprint arXiv:1411.0602* (2014).

[30] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. 2011. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical Programming* 127, 1 (2011), 3–30.

[31] Alexander Smirnov and Andrew Ponomarev. 2015. Locality-Sensitive Hashing for Distributed Privacy-Preserving Collaborative Filtering: An Approach and System Architecture. In *International Conference on Enterprise Information Systems*. Springer, 455–475.

[32] Xiaoyuan Su and Taghi M Khoshgoftaar. 2009. A survey of collaborative filtering techniques. *Advances in Artificial Intelligence* 2009 (2009), 4.

[33] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2016. Learning to hash for indexing big data-A survey. *Proc. IEEE* 104, 1 (2016), 34–57.

[34] Qifan Wang, Lingyun Ruan, Zhiwei Zhang, and Luo Si. 2013. Learning compact hashing codes for efficient tag completion and prediction. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*. ACM, 1789–1794.

[35] Yair Weiss, Antonio Torralba, and Rob Fergus. 2009. Spectral hashing. In *Advances in Neural Information Processing Systems*. 1753–1760.

[36] Hanwang Zhang, Fumin Shen, Wei Liu, Xiangnan He, Huanbo Luan, and Tat-Seng Chua. 2016. Discrete collaborative filtering. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, Vol. 16.

[37] Zhiwei Zhang, Qifan Wang, Lingyun Ruan, and Luo Si. 2014. Preference preserving hashing for efficient recommendation. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*. ACM, 183–192.

[38] Ke Zhou and Hongyuan Zha. 2012. Learning binary codes for collaborative filtering. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 498–506.