Candidate Generation with Binary Codes for Large-Scale Top-N Recommendation

Wang-Cheng Kang
University of California, San Diego
La Jolla, CA, USA
wckang@ucsd.edu

Julian McAuley
University of California, San Diego
La Jolla, CA, USA
jmcauley@ucsd.edu

ABSTRACT

Generating the Top-N recommendations from a large corpus is computationally expensive to perform at scale. Candidate generation and re-ranking based approaches are often adopted in industrial settings to alleviate efficiency problems. However it remains to be fully studied how well such schemes approximate complete rankings (or how many candidates are required to achieve a good approximation), or to develop systematic approaches to generate high-quality candidates efficiently. In this paper, we seek to investigate these questions via proposing a candidate generation and re-ranking based framework (CIGAR), which first learns a preference-preserving binary embedding for building a hash table to retrieve candidates, and then learns to re-rank the candidates using real-valued ranking models with a candidate-oriented objective. We perform a comprehensive study on several large-scale real-world datasets consisting of millions of users/items and hundreds of millions of interactions. Our results show that CIGAR significantly boosts the Top-N accuracy against state-of-the-art recommendation models, while reducing the query time by orders of magnitude. We hope that this work could draw more attention to the candidate generation problem in recommender systems.

ACM Reference Format:
Wang-Cheng Kang and Julian McAuley. 2019. Candidate Generation with Binary Codes for Large-Scale Top-N Recommendation. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357384.3357930

1 INTRODUCTION

Top-N recommendation is a fundamental task of a recommender system, which consists of generating a (short) list of N items that are highly likely to be interacted with (e.g. purchased, liked, etc.) by users. Precisely identifying these Top-N items from a large corpus is highly challenging, both from an accuracy and efficiency perspective. The vast number of items, both in terms of their variability and sparsity, makes the problem especially difficult when scaling up to real-world datasets. In particular, exhaustively searching through

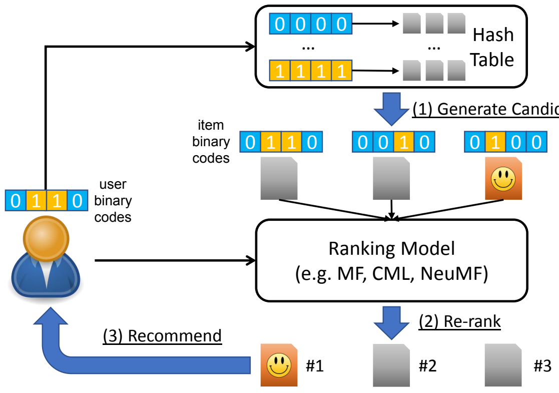


Figure 1: A simplified illustration showing the candidate generation and re-ranking procedures in our CIGAR framework. The binary codes and ranking model are both learned from user feedback.

all items to generate the Top-N ranking becomes intractable at scale due to its high latency.

Recommender systems have received significant attention with various models being proposed, though generally focused on the goal of achieving better accuracy [4, 8, 16, 36, 38]. For example, BPR-MF [32] adopts a conventional Matrix Factorization (MF) approach as its underlying preference model, CML [15] employs metric embeddings, TransRec [12] adopts translating vectors, and NeuMF [13] uses multi-layer perceptrons (MLP) to model user-item interactions.

As for the problem of latency/efficiency, a few works seek to accelerate the maximum inner product (MIP) search step (for MF-based models), via pruning or tree-based data structures [21, 31]. Such approaches are usually model-specific (e.g. they depend on the specific structure of an inner-product space), and thus are hard to generalize when trying to accelerate other models. Another line of work seeks to directly learn binary codes to estimate user-item interactions, and builds hash tables to accelerate retrieval time [24, 27, 44, 46–48]. While using binary codes can significantly reduce query time to constant or sublinear complexity, the accuracy of such models is still inferior to conventional (i.e., real-valued) models, as such models are highly constrained, and may lack sufficient flexibility when aiming to precisely rank the Top-N items.

As the vast majority of items will be irrelevant to most users at any given moment, candidate generation and re-ranking strategies have been adopted in industry where high efficiency is required. Such approaches first generate a small number of candidates in an efficient way, and then apply fine-grained re-ranking methods to obtain the final ranking. To achieve high efficiency, the candidate generation stage is often based on rules or heuristics. For example, *Youtube's* early recommender system treated users' recent actions as seeds, and searched among relevant videos in the co-visitation graphs with a heuristic relevance score [7]. *Pinterest* performs random walks (again using recent actions as seeds) on the pin-board graph to retrieve relevant candidates [10], and also considers other candidate sources based on various signals like annotations, content, etc. [26]. Recently, *Youtube* adopted deep neural networks (DNNs) to extract user embeddings from various features, and used an inner product function to estimate scores, such that candidate generation can be accelerated by maximum inner product search [5].

In this paper, we propose a novel candidate generation and re-ranking based framework called *CIGAR*. During the candidate generation stage, unlike existing work that adopts heuristics, or learns real-valued embeddings first and then adopts indexing techniques to accelerate, we propose to directly learn binary codes for both preference ranking and hash table lookup. During the re-ranking stage, we learn to re-rank candidates using existing ranking models with candidate-oriented sampling strategies. Figure 1 shows the procedure of generating recommendations using CIGAR.

Our main contributions are as follows:

- We propose a novel framework (**CIGAR**) which learns to generate candidates with binary codes, and re-ranks candidates with real-valued models. CIGAR thus exhibits both the efficiency of hashing and the accuracy of real-valued methods: binary codes are employed to estimate *coarse-grained* preference scores and efficiently retrieve candidates, while real-valued models are used for *fine-grained* re-ranking of a small number of candidates.
- We propose a new hashing-based method—**HashRec**—for learning binary codes with implicit feedback. HashRec is optimized via stochastic gradient descent, and can easily scale to large datasets. CIGAR adopts HashRec for fast candidate generation, as empirical results show that HashRec achieves superior performance compared to other hashing-based methods.
- We propose a candidate-oriented sampling strategy which encourages the models to focus on re-ranking candidates, rather than treating all items equally. With such a sampling scheme, CIGAR can significantly boost the accuracy of various exiting ranking models, including neural-based approaches.
- Comprehensive experiments are conducted on several large-scale datasets. We find that CIGAR outperforms the existing state-of-the-art models, including those that rank all items, while reducing the query time by orders of magnitude. Our results suggest that it is possible to achieve similar or better performance than existing approaches even when using only a small number of candidates.

2 BACKGROUND

In this section, we briefly review relevant background including representative ranking models for implicit feedback, and hashing-based models for efficient recommendation.

2.1 Preference Ranking Models

2.1.1 Recommendation with Implicit Feedback. In our paper, we focus on learning user preferences from implicit feedback (e.g. clicks, purchases, etc.). Specifically, we are given a user set $\mathcal{U}$ and an item set $\mathcal{I}$, such that the set $\mathcal{I}_u^+$ represents the items that user u has interacted with, while $\mathcal{I}_u^- = \mathcal{I} - \mathcal{I}_u^+$ represents unobserved interactions. Unobserved interactions are not necessarily negative, rather for the majority of such items the user may simply be unaware of them. To interpret such in-actions, weighted loss [17] and learning-to-rank [32] approaches have been proposed.

2.1.2 Bayesian Personalized Ranking (BPR). BPR [32] is a classic approach for learning preference ranking models from implicit feedback. The core idea is to rank observed actions (items) higher than unobserved items. BPR-MF is a popular variant that adopts conventional Matrix Factorization (MF) approaches as its underlying preference estimator:

$$s_{u,i} = \langle \mathbf{p}_u, \mathbf{q}_i \rangle, \tag{1}$$

where $\mathbf{p}_u, \mathbf{q}_i$ are k-dimensional embeddings. BPR seeks to optimize pairwise rankings by minimizing a contrastive objective:

$$\begin{aligned} &- \sum_{(u,i,j)\in\mathcal{D}} \ln \sigma(s_{u,i} - s_{u,j}) \\ &\mathcal{D} = \{(u,i,j) | u \in \mathcal{U} \wedge i \in \mathcal{I}^+ \wedge j \in \mathcal{I}^-\}. \end{aligned} \tag{2}$$

As enumerating all triplets in $\mathcal{D}$ is typically intractable, BPR-MF adopts stochastic gradient descent (SGD) to optimize the model. Namely, in each step of SGD, we dynamically sample a batch of triplets from $\mathcal{D}$. Also, an ℓ_2 regularization on user and item embeddings is adopted, which is crucial to alleviate overfitting.

2.1.3 Collaborative Metric Learning (CML). Conventional MF-based methods operate in inner product spaces, which are flexible but can easily overfit. To this end, CML [15] imposes the triangle inequality constraint, by adopting metric embeddings to represent users and items. Here the preference score is estimated by the negative ℓ_2 distance:

$$s_{u,i} = -\|\mathbf{p}_u - \mathbf{q}_i\|_2. \tag{3}$$

CML adopts a hinge loss to optimize pairwise rankings. A significant benefit of CML is that retrieval can be accelerated by efficient nearest neighbor search, which has been heavily studied.

2.1.4 Neural Matrix Factorization (NeuMF). To estimate more complex and non-linear preference scores, NeuMF [13] adopts multi-layer perceptrons for modeling interactions:

$$s_{u,i} = \mathbf{w}^T \begin{bmatrix} \mathbf{p}_u^{(1)} \odot \mathbf{q}_i^{(1)} \\ \text{MLP}(\mathbf{p}_u^{(2)}, \mathbf{q}_i^{(2)}) \end{bmatrix}, \tag{4}$$

where $\odot$ is the element-wise product, 'MLP' extracts a vector from user and item embeddings, and $\mathbf{w}$ is used to project the concatenated vector to the final score. Essentially NeuMF combines generalized matrix factorization (GMF) and MLPs. Due to the complexity of the scoring function, the retrieval process is generally hard to accelerate for NeuMF.

2.2 Hashing-based Recommendation

To achieve efficient recommendation, various hashing-based models have been proposed. These methods use binary representations to represent users and items, and the retrieval time can be reduced to constant or sublinear time by appropriate use of a hash table. We

Table 1: Notation.

Notation	Description
$\mathcal{U}, \mathcal{I}$	user and item set
$r \in \mathbb{N}$	binary embedding length (#bits)
$k \in \mathbb{N}$	real-valued embedding size
$c \in \mathbb{N}$	number of candidates for re-ranking
$\widetilde{\mathbf{b}}_u, \widetilde{\mathbf{d}}_i \in \mathbb{R}^r$	auxiliary embeddings for user u and item i
$\mathbf{b}_u, \mathbf{d}_i \in \{-1, 1\}^r$	binary embeddings for user u and item i
$\mathbf{p}_u, \mathbf{q}_i \in \mathbb{R}^k$	real embeddings for user u and item i
$m \in \mathbb{N}$	the number of substrings in MIH
$h \in \mathbb{R}$	the sampling ratio in eq. 10

briefly introduce the Hamming Space and two relevant hashing-based recommendation method.

2.2.1 Hamming Space. A Hamming space contains 2^r binary strings with length r. Binary codes can be efficiently stored and computed in modern systems. In this paper we use binary codes $\mathbf{b}_u, \mathbf{d}_i \in \{-1, 1\}^r$ to represent users and items.[1] The negative Hamming distance measures the similarity between two binary strings:

$$\begin{aligned} s_H(\mathbf{b}_u, \mathbf{d}_i) &= \sum_{z=1}^{r} \mathbb{I}(b_{u,z} = d_{i,z}) \\ &= \frac{1}{2}\left(\sum_{z=1}^{r} \mathbb{I}(b_{u,z} = d_{i,z}) + r - \sum_{z=1}^{r} \mathbb{I}(b_{u,z} \neq d_{i,z})\right) \\ &= const + \frac{1}{2}\langle \mathbf{b}_u, \mathbf{d}_i \rangle, \end{aligned} \tag{5}$$

where $\mathbb{I}(\cdot)$ is the indicator function. This provides a convenient way to formulate the problem with the inner product.

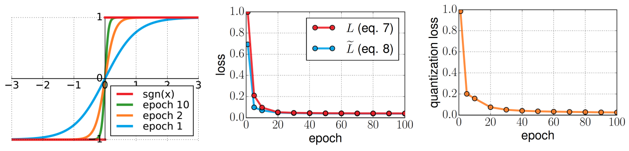
2.2.2 Discrete Collaborative Filtering (DCF). DCF [44] is a representative method that estimates observed ratings (scaled to $[-r, r]$) using $\langle \mathbf{b}_u, \mathbf{d}_i \rangle$. Additional constraints of bit balance and bit uncorrelation are adopted to learn efficient binary codes. DCF introduces real-valued auxiliary variables, and adopts an optimization strategy consisting of alternating sub-problems with closed-form solutions.

2.2.3 Discrete Personalized Ranking (DPR). To our knowledge, DPR [46] is the only hashing-based method designed for implicit feedback. DPR considers triplets $\mathcal{D}$ as in BPR, and optimizes rankings using a squared loss. DPR also optimizes sub-problems with closed-form solutions. However, the solutions to these sub-problems rely on computing all triplets in $\mathcal{D}$, which makes optimization hard to scale to large datasets.

3 CIGAR: LEARNING TO GENERATE CANDIDATES AND RE-RANK

In this section, we introduce CIGAR, a candidate generation and re-ranking based framework. We propose a new method HashRec that learns binary embeddings for users and items. CIGAR leverages the binary codes generated by HashRec, to construct hash tables for fast candidate generation. Finally, CIGAR learns to re-rank candidates via real-valued ranking models with the proposed sampling strategy. Our notation is summarised in Table 1.

[1]We use {-1,1} instead of {0,1} for convenience of formulations, though in practice we can convert to binary codes (i.e., {0,1}) when storing them.



(a) $\mathrm{sgn}(x)$ vs. $\tanh(\beta x)$ (b) Training loss (c) Quantization loss

Figure 2: Training curves on MovieLens-20M. Figure (a) plots $\mathrm{sgn}(x)$ **and its approximation** $\tanh(\beta x)$**. Figure (b) plots the desired loss** L **and surrogate loss** $\widetilde{L}$ **through training. Figure (c) shows the quantization error (measured via mean squared distances) between** $\mathrm{sgn}(x)$ **and** $\tanh(\beta x)$**.**

3.1 Learning Preference-preserving Binary Codes

We use binary codes $\mathbf{b}_u, \mathbf{d}_i \in \{-1, 1\}^r$ to represent users and items, and estimate interactions between them via the Hamming distance. We seek to learn preference-preserving binary codes such that similar binary codes (i.e., low Hamming distance) indicate high preference scores. For convenience, we use the conventional inner product (the connection is shown in eq. 5) in our formulation:

$$s_{u,i} = \langle \mathbf{b}_u, \mathbf{d}_i \rangle. \tag{6}$$

In implicit feedback settings, we seek to rank observed interactions (u,i) higher than unobserved interactions. To achieve this, we employ the classic BPR [32] loss to learn our binary codes. However, directly optimizing such binary codes is generally NP-Hard [42]. Hence, we introduce auxiliary real-valued embeddings $\widetilde{\mathbf{b}}_u, \widetilde{\mathbf{d}}_i \in \mathbb{R}^r$ as used by other learning-to-hash approaches [44]. Thus our objective is equivalent to:

$$L = -\sum_{(u,i,j)\in\mathcal{D}} \ln \sigma_\alpha \left(\langle \mathrm{sgn}(\widetilde{\mathbf{b}}_u), \mathrm{sgn}(\widetilde{\mathbf{d}}_i) \rangle - \langle \mathrm{sgn}(\widetilde{\mathbf{b}}_u), \mathrm{sgn}(\widetilde{\mathbf{d}}_j) \rangle \right), \tag{7}$$

where $\mathrm{sgn}(x) = 1$ if $x \geq 0$ (-1 otherwise), and $\sigma_\alpha(x) = \sigma(\alpha x) = 1/(1 + \exp(-\alpha x))$. As the inner product between binary codes can be large (i.e., $\pm r$), we set $\alpha < 1$ to reduce the saturation zone of the sigmoid function. Inspired by a recent study for image hashing [3, 18], we seek to optimize the problem by approximating the $\mathrm{sgn}(\cdot)$ function:

$$\mathrm{sgn}(x) = \lim_{\beta\to\infty} \tanh(\beta x),$$

where β is a hyper-parameter that increases during training. With this approximation, the objective becomes:

$$\begin{aligned} \widetilde{L} = -\sum_{(u,i,j)\in\mathcal{D}} \ln \sigma_\alpha \Big(&\langle \tanh(\beta\widetilde{\mathbf{b}}_u), \tanh(\beta\widetilde{\mathbf{d}}_i) \rangle - \\ &\langle \tanh(\beta\widetilde{\mathbf{b}}_u), \tanh(\beta\widetilde{\mathbf{d}}_j) \rangle \Big). \end{aligned} \tag{8}$$

As shown in Figure 2, when we optimize the surrogate loss $\widetilde{L}$, the desired loss L is also minimized consistently. Also we can see that the quantization loss (i.e., the mean squared distances between $\mathrm{sgn}(x)$ and $\tanh(\beta x)$) drops significantly throughout the training process. Note that we also employ ℓ_2-regularization on embeddings $\widetilde{\mathbf{b}}_u$ and $\widetilde{\mathbf{d}}_i$, as in BPR.

We name this method **HashRec**; the complete algorithm is given in Algorithm 1.

Algorithm 1 Optimization in HashRec

```
**Input:** training data $\mathcal{D}$, code length $r$, regularization coefficient $\lambda$
Initialize embeddings $\widetilde{\mathbf{B}} \in \mathbb{R}^{|\mathcal{U}|\times r}$ and $\widetilde{\mathbf{D}} \in \mathbb{R}^{|\mathcal{I}|\times r}$ (at random)
for $epoch = 1 \rightarrow num\_epochs$ do
    $\beta \leftarrow \sqrt{10 * (epoch - 1)}$
    for $iter = 1 \rightarrow num\_iterations$ do
        Sample a batch of triplets a from $\mathcal{D}$
        Optimize loss (8) by updating $\widetilde{\mathbf{b}}_u$ and $\widetilde{\mathbf{d}}_i$ using the Adam [19] optimizer
Obtain binary codes by $\mathbf{b}_u \leftarrow \text{sgn}(\widetilde{\mathbf{b}}_u)$ and $\mathbf{d}_i \leftarrow \text{sgn}(\widetilde{\mathbf{d}}_i)$
**Output:** $\mathbf{B} \in \{-1, 1\}^{|\mathcal{U}|\times r}$, $\mathbf{D} \in \{-1, 1\}^{|\mathcal{I}|\times r}$
```

3.2 Building Multi-Index Hash Tables

Using binary codes to represent users and items can yield significant benefits in terms of storage cost and retrieval speed. For example, in our experiments, HashRec achieves satisfactory accuracy with r=64 bits, which is equivalent in space to only 4 single-precision floating-point numbers (i.e., float16). Moreover, computing architectures are amenable to calculating the Hamming distance of binary codes.[2] In other words, performing exhaustive search with binary codes is much faster (albeit by a constant factor) compared to real-valued embeddings. However, using exhaustive search inevitably leads to linear time complexity (in $|\mathcal{I}|$), which still scales poorly.

To scale to large, real-world datasets, we seek to build hash tables to index all items according to their binary codes, such that we can perform hash table lookup to retrieve and recommend items for a given user. Specifically, for a query code $\mathbf{b}_u$, we retrieve items from buckets within a small radius l (i.e., $d_H(\mathbf{b}_u, \mathbf{d}_i) \leq l$). Hence the returned items have low Hamming distances (i.e., high preference scores) compared to the query codes, and search can be done in constant time. However, for large code lengths the number of buckets grows exponentially, and furthermore such an approach may return zero items as nearby buckets will frequently be empty due to dataset sparsity.

Hence we employ Multi-Index Hashing (MIH) [30] as our indexing data structure. The core idea of MIH is to split binary codes to m substrings and index them by m hash tables. When we retrieve items within Hamming radius l, we first retrieve items in each hash table with radius $\lfloor \frac{l}{m} \rfloor$, and then sort the retrieved items based on their Hamming distances with the full binary codes. It can be guaranteed that such an approach can retrieve the desired items (i.e., within Hamming radius l), and that the search time is sub-linear in the number of items [30].

Since we are interested in generating fixed-length (Top-N) rankings, we seek to retrieve c items as candidates, instead of considering Hamming radii. MIH proposes an adaptive solution that gradually increases the radius l until enough items are retrieved (i.e., at least c). Empirically we found that the query time of MIH is extremely fast and grows slowly with the number of items. The pseudo-code for constructing and retrieving items in MIH, and more information about this process is described in the appendix.

[2]The Hamming distance can be efficiently calculated by two instructions: XOR and POPCNT (count the number of bits set to 1).

3.3 Candidate-oriented Re-ranking

So far we have learned preference-preserving binary codes for users and items, and constructed hash tables to efficiently retrieve items for users. However, as observed in previous hashing-based methods, generating recommendations purely using binary codes leads to inferior accuracy compared with conventional real-valued ranking models. To achieve satisfactory performance in terms of both accuracy and efficiency, we propose to use the retrieved items as candidates, and adopt sophisticated ranking models to refine the results. As the preference ranking problem has been heavily studied [13, 15, 32], we employ existing models to study the effect of the CIGAR framework, and propose a candidate-oriented sampling strategy to further boost accuracy.

A straightforward approach would be to adopt 'off-the-shelf' ranking models (e.g. BPR-MF) for re-ranking. However, we argue that such an approach is sub-optimal as existing models are typically trained to produce rankings for all items, while our re-ranking models only rank the c generated candidates. Moreover, the retrieved candidates are often 'difficult' items (i.e., items that are hard for ranking models to discriminate) or at the very least are not a uniform sample of items. Hence, it might be better to train ranking models such that they are focused on the re-ranking objective. In this section, we introduce our candidate-oriented sampling strategy, and show how to apply it to existing ranking models in general.

The loss functions of preference ranking models can generally be described as[3]:

$$\min \sum_{(u,i,j)\in\mathcal{D}} \mathcal{L}(u, i, j). \tag{9}$$

The choice of $\mathcal{L}$ is flexible, for instance, BPR uses $\ln \sigma(\cdot)$, and CML adopts the margin loss. To make the model focus more on learning to re-rank the candidates, we propose a candidate-oriented sampling strategy, which substitutes $\mathcal{D}$ with

$$\mathcal{D}^+ = \begin{cases} \text{sample from } \mathcal{D}, & \text{with probability } 1 - h \\ \text{sample from } C, & \text{with probability } h \end{cases}, \tag{10}$$

where C=$\{(u, i, j)|u \in \mathcal{U} \wedge i \in \mathcal{I}_u^+ \wedge j \in \mathcal{I}_u^- \cap C_u\}$, C_u contains c candidates for user u generated by hashing, and h controls the probability ratio. Note that the sampling is equivalent to assigning larger weights to the candidates (for h>0). We empirically find that the best performance is obtained with 0<h<1; when h=0, the model is not aware of the candidates that need to be ranked, while h=1 may lead to overfitting due to the limited number of samples.

As for constructing C, one approach is online generation, as we did for $\mathcal{D}$. Namely, in each step of SGD, we sample a batch of users and obtain candidates by hashtable lookup. Another approach is to pre-compute and store all candidates. Both approaches are practical, though we adopt the latter for better time efficiency.

Finally, taking BPR-MF as an example, the candidate-oriented re-ranking model is trained with the following objective:

$$-\sum_{(u,i,j)\in\mathcal{D}^+} \ln \sigma\left(\langle \mathbf{p}_u, \mathbf{q}_i\rangle - \langle \mathbf{p}_u, \mathbf{q}_j\rangle\right). \tag{11}$$

We denote this model as BPR-MF$^+$, to distinguish against the vanilla model. CML$^+$ and NeuMF$^+$ (etc.) are denoted in the same way.

[3]For pairwise learning based methods (e.g. NeuMF), we have $\mathcal{L}(u, i, j) = \mathcal{L}^+(u, i) + \mathcal{L}^-(u, j)$.

3.4 Summary

To summarize, the training process of CIGAR consists of: (1) Learning preference-preserving binary codes $\mathbf{b}_u$ and $\mathbf{d}_i$ using HashRec (Algorithm 1); (2) Constructing Multi-Index Hash tables to index all items; (3) Training a ranking model (e.g. BPR-MF$^+$) for re-ranking candidates (i.e., using a candidate-oriented sampling strategy). We adopt SGD to learn binary codes as well as our re-ranking model, such that optimization easily scales to large datasets.

During testing, for a given user u, we first retrieve c candidates via hashtable lookup, then we adopt a linear scan to calculate their scores estimated by the re-ranking model, and finally return the Top-N items with the largest scores. Using candidates generated by hash tables significantly reduces the time complexity from linear (i.e., exhaustive search) to sub-linear, and in practice is over 1,000 times faster for the largest datasets in our experiments.

For hyper-parameter selection, by default, we set the number of candidates c=200, the number of bits r=64, the scaling factor α=$10/r$, and sampling ratio h=0.5. The ℓ_2 regularizer λ is tuned via cross-validation. In MIH, the number of substrings m is manually set depending on dataset size. For example, we set m=4 for datasets with millions of items. Further details are included in the appendix.

4 EXPERIMENTS

We conduct comprehensive experiments to answer the following research questions:

RQ1: Does CIGAR achieve similar or better Top-N accuracy compared with state-of-the-art models that perform exhaustive rankings (i.e., which rank all items)?

RQ2: Does CIGAR accelerate the retrieval time of inner-product-, metric-, or neural-based models on large-scale datasets?

RQ3: Does HashRec outperform alternative hashing-based approaches? Do candidate-oriented sampling strategies (e.g. BPR-MF$^+$) help?

RQ4: What is the influence of key hyper-parameters in CIGAR?

The code and data processing script are available at https://github.com/kang205/CIGAR.

4.1 Datasets

We evaluate our methods on four public benchmark datasets. A proprietary dataset from *Flipkart* is also employed to test the scalability of our approach on a representative industrial dataset. The datasets vary significantly in domains, sparsity, and number of users/items; dataset statistics are shown in Table 2. We consider the following datasets:

- **MovieLens**[4] A widely used benchmark dataset for evaluating collaborative filtering algorithms [11]. We use the largest version that includes 20 million user ratings. We treat all ratings as implicit feedback instances (since we are trying to predict whether users will *interact* with items, rather than their ratings).
- **Amazon**[5] A series of datasets introduced in [29], including large corpora of product reviews crawled from *Amazon.com*. Top-level product categories on *Amazon* are treated as separate datasets, and we use the largest category 'books.' All reviews are treated as implicit feedback.
- **Yelp**[6] Released by *Yelp*, containing various metadata about businesses (e.g. location, category, opening hours) as well as user reviews. We use the Round-12 version, and regard all review actions as implicit feedback.
- **Goodreads.**[7] A recently introduced large dataset containing book metadata and user actions (e.g. shelve, read, rate) [39]. We treat the most abundant action ('shelve') as implicit feedback. As shown in [39], the dataset is dominated by a few popular items (e.g. over 1/3 of users added *Harry Potter #1* to their shelves), such that always recommending the most popular books achieves high Top-N accuracy; we ignore such outliers by discarding the 0.1% of most popular books.
- **Flipkart** A large dataset of user sessions from *Flipkart.com*, a large online electronics and fashion retailer in India. The recorded actions include 'click,' 'purchase,' and 'add to wishlist.' Data was crawled over November 2018. We treat all actions as implicit feedback.

Table 2: Dataset statistics (after preprocessing)

Dataset	#Items	#Users	#Actions	% Density
MovieLens-20M	18K	138K	20M	0.81
Yelp	103K	244K	3.7M	0.015
Amazon Books	368K	604K	8.9M	0.004
GoodReads	1.6M	759K	167M	0.014
Flipkart	2.9M	9.0M	274M	0.001

For all datasets, we take the k-core of the graph to ensure that all users and items have at least 5 interactions. Following [4, 32], we adopt a *leave-one-out* strategy for data partitioning: for each user, we randomly select two actions, put them into a validation set and test set respectively, and use all remaining actions for training.

4.2 Evaluation Protocol

We adopt two common Top-N metrics: Hit Rate (HR) and Mean Reciprocal Rank (MRR), to evaluate recommendation performance [4, 12]. HR@N counts the fraction of times that the single left-out item (i.e., the item in the test set) is ranked among the top N items, while MRR@N is a position-aware metric which assigns larger weights to higher positions (i.e., $1/i$ for the i-th position). Note that since we only have one test item for each user, HR@N is equivalent to Recall@N, and is proportional to Precision@N. Following [4], we set N to 10 by default.

4.3 Baselines

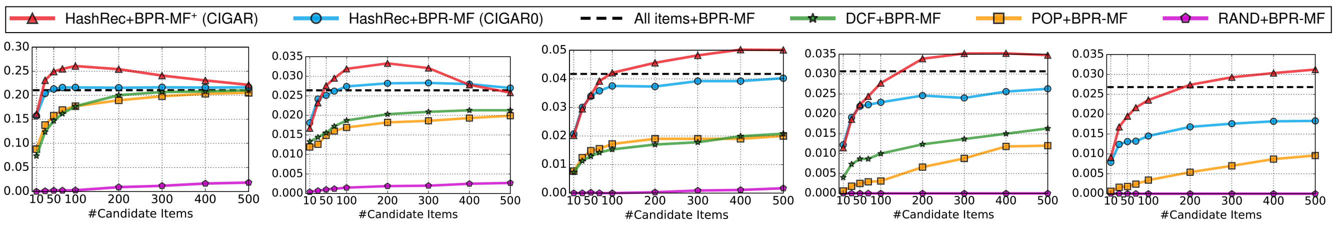
We consider three representative recommendation models that estimate user-item preference scores with inner-products, Euclidean distances, and neural networks:

- **Bayesian Personalized Ranking (BPR-MF) [32]** A classic model that seeks to optimize a pairwise ranking loss. We employ MF as its preference predictor as shown in eq. 1. When recommending items, a maximum inner product search is needed.

[4]https://grouplens.org/datasets/movielens/20m/
[5]http://jmcauley.ucsd.edu/data/amazon/index.html
[6]https://www.yelp.com/dataset/challenge
[7]https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home

Table 3: Recommendation performance. The best performing method in each row is boldfaced, and the second best method in each row is underlined. All the numbers are shown in percentage. - means the training fails due to lack of memory.

Dataset	Metric	(a-1) POP	(a-2) BPR-B	(a-3) DCF	(a-4) HashRec	Metric	(b-1) BPR-MF	(b-2) CML	(b-3) NeuMF	CIGAR0 HashRec+BPR-MF	CIGAR HashRec+BPR-MF^{+}	Improv. %
ML-20M	HR@10	8.15	11.85	6.90	**15.72**	HR@10	21.05	21.41	18.43	21.56	**25.42**	18.7
	HR@200	42.35	51.88	43.31	**62.96**	MRR@10	8.82	8.92	7.01	8.78	**11.46**	28.4
Yelp	HR@10	1.03	0.21	1.30	**1.39**	HR@10	2.64	2.12	1.70	2.82	**3.33**	26.1
	HR@200	7.82	3.38	10.19	**18.49**	MRR@10	0.88	0.69	0.57	0.97	**1.22**	38.6
Amazon	HR@10	0.74	1.26	0.91	**2.08**	HR@10	4.17	3.47	1.59	3.73	**4.56**	9.4
	HR@200	5.14	6.56	6.45	**11.69**	MRR@10	1.73	1.41	0.67	1.91	**2.23**	28.9
GoodReads	HR@10	0.06	0.44	0.40	**1.19**	HR@10	3.07	3.20	2.20	2.46	**3.39**	5.9
	HR@200	1.61	2.86	3.99	**8.10**	MRR@10	1.42	1.26	0.91	1.15	**1.73**	21.8
Flipkart	HR@10	0.05	0.34	-	**0.92**	HR@10	2.68	0.81	-	1.68	**2.74**	2.2
	HR@200	1.24	1.92	-	**6.13**	MRR@10	1.03	0.30	-	0.64	**1.19**	13.6



(a) ML-20M **(b) Yelp** **(c) Amazon Books** **(d) Goodreads** **(e) Flipkart**

Figure 3: Effect of the number of candidates.

- **Collaborative Metric Learning (CML) [15]** CML represents users and items in a metric space, and measures their compatibility via the Euclidean distance (as shown in eq. 3). The recommended items for a user can be retrieved via nearest neighbor search.
- **Neural Matrix Factorization (NeuMF) [13]** NeuMF models non-linear interactions between user and item embeddings via a multi-layer perceptron (MLP). A generalized matrix factorization (GMF) term is also included in the model as in eq. 4.

We also compare HashRec with various hashing-based recommendation approaches:

- **POP** A naïve popularity-oriented baseline that simply ranks items by their global popularity.
- **BPR-B** A simple baseline that directly quantizes embeddings from BPR-MF (i.e., applying sgn(x) to the embeddings).
- **Discrete Collaborative Filtering (DCF) [44]** DCF learns binary embeddings to estimate observed ratings. To adapt it to the implicit feedback setting, we treat all actions as having value 1 and randomly sample 100 unobserved items (for each user) with value 0.
- **Discrete Personalized Ranking (DPR) [46]** DPR is a hashing-based method designed for optimizing ranking with implicit feedback. However, due to its high training complexity, we only compare against this approach on MovieLens-1M.

The comparison against other hashing-based methods is omitted, as they are either content-based [24, 27, 48], designed for explicit feedback [47], or outperformed by our baselines [28].

Finally, our candidate generation and re-ranking based framework **CIGAR**, which first retrieves $c = 200$ candidates from the Multi-Index Hash (MIH) table, and adopts ranking models to re-rank the candidates to obtain final recommendations. By default, CIGAR employs HashRec to learn binary user/item embeddings, and BPR-MF^{+} as the re-ranking model. The effect of CIGAR with different candidate generation methods, re-ranking models, and hyper-parameters are also studied in the experiments. More details on hyperparameter tuning are included in the appendix.

4.4 Recommendation Performance

Table 3 shows Top-N recommendation accuracy on all datasets. Columns (a-1) to (a-4) contain 'efficient' recommendation methods (i.e., based on popularity or hashing), while (b-1) to (b-3) represent real-valued ranking models. For hashing-based methods, we use HR@200 to evaluate the performance of candidate generation (i.e., whether the desired item appears in the 200 candidates).

Not surprisingly, there is a clear gap between hashing-based methods and real-valued methods in terms of HR@10, which confirms that using binary embeddings alone makes it difficult to identify the fine-grained Top-10 ranking due to the compactness of the binary representations. However, we find that the HR@200 of HashRec (and DCF) is significantly higher than the HR@10 of (b-1) to (b-3), which suggests the potential of using hashing-based methods to generate coarse-grained candidates, as the HR@200 during the candidate generation stage is an upper bound for the Top-10 performance (e.g. if we have a perfect re-ranking model, the HR@10 would be equal to the HR@200) using the CIGAR framework.

Table 3 shows that HashRec significantly outperforms hashing-based baselines, presumably due to the tanh(·) approximation and the use of the advanced *Adam* optimizer. Hence, we choose HashRec as the candidate generation method in CIGAR by default. Finally, we see that CIGAR (with HashRec and BPR-MF$^+$) outperforms state-of-the-art recommendation approaches. Note that CIGAR only ranks 200 candidates (generated by HashRec), while BPR-MF, CML and NeuMF rank all items to obtain the Top-10 results. This suggests that only considering a small number of high-quality candidates is sufficient to achieve satisfactory performance.

Comparison against DPR We perform a comparison with DPR [46] on the smaller dataset MovieLens-1M as the DPR is hard to scale to other datasets we considered. Following the same data filtering, partitioning, and evaluation scheme, DPR achives an HR@10 of 8.9%, HR@200 of 55.7%. In comparison, HashRec's HR@10 is 13.5%, and HR@200 is 64.6%. This shows that HashRec outperforms DPR which is also designed for implicit recommendation.

4.5 Effect of the Number of Candidates

Figure 3 shows the HR@10 of various approaches with different numbers of candidates. We can observe the effect of different candidate generation methods by comparing HashRec, DCF, POP, and RAND with a fixed ranking approach (BPR-MF). CIGAR0 (HashRec+BPR-MF) clearly outperforms alternate approaches, and achieves satisfactory performance (similar to All Items+BPR-MF) on the first three datasets. For larger datasets, more bits and more candidates might be helpful to boost the performance (see sec. 4.8).

However, CIGAR0 merely approximates the performance of All Items+BPR-MF. As we pointed out in section 3.3, this approach is suboptimal as the vanilla BPR-MF is trained to rank all items, whereas we need to rank a small number of 'hard' candidates in the CIGAR framework. By adopting the candidate-oriented sampling strategy to train a BPR-MF model focusing on ranking candidates, we see that CIGAR achieves significant improvements over All Items+BPR-MF. This confirms that the proposed candidate-oriented re-ranking strategy is crucial in helping CIGAR to achieve better performance than the original ranking model.

Note that CIGAR is trained to re-rank the c=200 candidates. This may cause the performance drop when ranking more candidates on small datasets like *ML-20M* and *Yelp*.

4.6 Effects of Candidate-oriented Re-ranking

In previous sections, we have shown the performance of CIGAR with different candidate generation methods (e.g. HashRec, DCF, POP). Since CIGAR is a general framework, in this section we examine the performance of CIGAR using CML and NeuMF as the re-ranking model (BPR-MF is omitted here, as results are included in Figure 3), so as to investigate whether the candidate-oriented sampling strategy is helpful in general.

Table 4 lists the performance (HR@10) of CIGAR using CML and NeuMF as its ranking model. Due to the high quality of the 200 candidates, HashRec+CML and HashRec+NeuMF can achieve comparable performance compared to rank all items with the same model. Moreover, we can consistently boost the perfomance via retraining the model with the candidate-oriented sampling strategy (i.e., CML$^+$ and NeuMF$^+$), which shows the mixed sampling is the key factor for outperforming the vanilla models with exhaustive searches (refer to section 4.8 for more analysis).

Table 4: Effects of the candidate-oriented re-ranking sampling with different ranking models. ↑ indicates better performance than ranking all items with the same model.

Approach	ML-20M	Yelp	Amazon	Goodreads
All items + CML	21.41	2.12	3.47	3.20
HashRec + CML	21.27	2.33↑	3.34	2.90
HashRec + CML$^+$	23.62↑	3.19↑	4.22↑	3.31↑
All items + NeuMF	18.43	1.70	1.59	2.20
HashRec + NeuMF	18.29	2.17↑	2.70↑	1.96
HashRec + NeuMF$^+$	20.83↑	2.37↑	2.78↑	2.74↑

4.7 Recommendation Efficiency

Efficiently retrieving the Top-N recommended items for users is important in real-world, interactive recommender systems. In Table 5, we compare CIGAR with alternative retrieval approaches for different ranking models. For all ranking models, a linear scan can be adopted to retrieve the top-10 items. BPR-MF is based on inner products, hence we adopt the MIP (Maximum Innder Product) Tree [31] to accelerate search speed. As CML requires a nearest neighbor search in a Euclidean space, we employ the classic KD-Tree and Ball Tree for retrieval. Since NeuMF utilizes neural networks to estimate preference scores, we use a GPU to accelerate the scan.

Table 5: Running times for recommending the Top-10 items to 1,000 users.

Model	Retrieval	Wall Clock Time(s)			
	Approach	Yelp	Amazon	Goodreads	Flipkart
#Items		0.1M	0.4M	1.6M	2.9M
BPR-MF	Linear Scan	109.0	375.7	1623.6	3076.4
	MIP Tree [31]	52.8	559.5	26.5	300.8
	CIGAR	1.2	1.5	1.6	1.9
CML	Linear Scan	375.8	1439.9	5972.5	12367.1
	KD Tree	18.2	40.4	162.2	169.1
	Ball Tree	15.4	46.3	210.7	227.8
	CIGAR	1.7	2.0	2.1	2.3
NeuMF	Parallel Scan (GPU)	21.4	76.1	332.4	-
	CIGAR	1.8	2.1	2.3	-

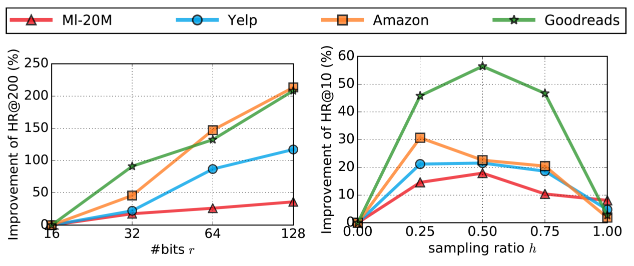
On the largest dataset (Flipkart), CIGAR is at least 1,000 times faster than linear scan for all models, and around 100 times faster than tree-based methods. Furthermore, compared to other methods the retrieval time of CIGAR increases very slowly with the number of items. Taking CML as an example, from Yelp to Flipkart, the query time for linear scan, KD Tree, and CIGAR increases by around 30x, 9x, and 1.4x (respectively). The fast retrieval speed of CIGAR is mainly due to the efficiency of hashtable lookup, and the small number of high-quality candidates for re-ranking.

Unlike KD-Trees, which are specifically designed for accelerating search in models based in Euclidean spaces, CIGAR is a model-agnostic approach that can efficiently accelerate the retrieval time of almost any ranking model, including neural models. Meanwhile, as shown previously, CIGAR can achieve better accuracy compared with models that rank all items. We note that MIP Tree performs extremely well on Goodreads. One possible reason is that a few learned vectors have large length[8], and hence the MIP Tree can quickly rule out most items.

Our approach is also efficient for training, for example, the whole training process of HashRec + BPR-MF$^+$ on the largest public dataset *Goodreads* can be finished in 3 hours (CPU only).

4.8 Hyper-parameter Study

In Figure 4, we show the effects of two important hyper-parameters: the number of bits r used in HashRec and the sample ratio h for learning the re-ranking model. For the number of bits, we can clearly observe that more bits leads to better performance. The improvement is generally more significant on larger datasets. For the sample ratio h, the best value is around 0.5 for most datasets, thus we choose h=0.5 by default. When h=1.0, the model seems to overfit due to limited data (i.e., a small number of candidates), and performance degrades. When h=0, the model is reduced to the original version which uniformly samples across all items. This again verifies the effect of the proposed candidate-oriented sampling strategy, as it significantly boosts performance compared to uniform sampling.



(a) #bits r **(b) Sampling ratio h**

Figure 4: Effects of the Hyper-parameters. (a) improvement of HashRec with more than 16 bits; (b) performance with different sampling ratios.

5 RELATED WORK

Efficient Collaborative Filtering Hashing-based methods have previously been adopted to accelerate retrieval for recommendation. Early methods adopt 'two-step' approaches, which first solve a relaxed optimization problem (i.e., binary codes are relaxed to real values), and obtain binary codes by quantization [28]. Such approaches often lead to a large quantization error, as learned embeddings in the first stage may be not ideal for quantization. To this end, recent methods jointly optimize the recommendation problem (e.g. rating prediction) and the quantization loss [44, 46], leading to better performance compared to two-step methods. For comparison, we highlight the main differences between our method and existing methods (e.g. DCF[44] and DPR [46]) as follows: (1) existing models are often optimized by closed-form solutions involving various matrix operations, whereas HashRec uses a more scalable SGD-based approach; (2) we do not impose bit balance or decorrelation constraints as in DCF and DPR, however in practice we did not observe any performance degradation in terms of accuracy and efficiency; (3) we use $\tanh(\beta x)$ to gradually close the gap between binary codes and continuous embeddings during training, which shows effective empirical approximation. To the best of our knowledge, HashRec is the first scalable hashing-based method for implicit recommendation.

Another line of work seeks to accelerate the retrieval process of existing models. For example, approximate nearest neighbor (ANN) search has been heavily studied [6, 43], which can be used to accelerate metric-based models like CML [15]. Recently, a few approaches have sought to accelerate the maximum inner product search (MIPS) operation for inner product based models [21, 31, 35]. However, these approaches are generally model-dependant, and can not easily be adapted to other models. For example, it is difficult to accelerate search for models that use complex scoring functions (e.g. MLP-based models such as NeuMF [13]). In comparison, CIGAR is a model-agnostic approach that can generally expedite search within any ranking model, as we only require the ranking model to scan a short list of candidates.

Inspired by knowledge distillation [14], a recent approach seeks to learn compact models (i.e., with smaller embedding sizes) while maintaining recommendation accuracy [37]. However, the retrieval complexity is still linear in the number of items.

Candidate Generation and Re-ranking To build real-time recommender systems, candidate generation has been adopted in industrial settings like *Youtube*'s video recomendation [5, 7], *Pinterest*'s related pin recommendation [10, 26], *Linkedin*'s job recommendation [2], and *Taobao*'s product recommendation [49]. Such industrial approaches often adopt heuristic rules, similarity measurements, and feature engineering specially designed for their own platforms. A closer work to our approach is *Youtube*'s method [5] which learns two models for candidate generation and re-ranking. The scoring function in the candidate generation model is the inner product of user and item embeddings, and thus can be accelerated by maximum inner product search via hashing- or tree-based methods. In comparison, our candidate generation method (HashRec) directly learns binary codes for representing preferences as well as building hash tables. To our knowledge, this is the first attempt to adopt hash code learning techniques for candidate generation.

Other than recommender systems, candidate generation has also been adopted in document retrieval [1], and NLP tasks[45].

Learning to Hash Unlike conventional data structures where hash functions might be designed (or learned) so as to reduce conflicts [9, 20], we consider similarity-preserving hashing that seeks to map high-dimensional dense vectors to a low-dimensional Hamming space while preserving similarity. Such approaches are often used to accelerate approximate nearest neighbor (ANN) search and reduce storage cost. A representative example is Locality Sensitive Hashing (LSH) [6] which uses random projections as the hash functions. A few seminal works [33, 42] propose to learn hash functions from data, which is generally more effective than LSH. Recent work focuses on improving the performance, (e.g.) by using better quantization strategies, and adopting DNNs as hash functions [40, 41]. Such approaches have been adopted for fast retrieval of various content including images [23, 34], documents [22], videos [25], and products (e.g. DCF [44] and HashRec). HashRec directly learns binary codes for users and items, which is essentially a hash function

[8]As shown in the appendix, the regularization coefficient is set to 0 for Goodreads, which may cause such a phenomenon.

projecting one-hot vectors into the Hamming Space. We plan to learn hash functions (e.g. based on DNNs) to map user and item features to binary codes as future work, such that we can adapt to new users and items, which may alleviate cold-start problems.

6 CONCLUSION

We presented new techniques for candidate generation, a critical (but somewhat overlooked) subroutine in recommender systems that seek to efficiently generate Top-N recommendations. We sought to bridge the gap between two existing modalities of research: methods that advance the state-of-the-art for Top-N recommendation, but are generally inefficient when trying to produce a final ranking; and methods based on binary codes, which are efficient in both time and space but fall short of the state-of-the-art in terms of ranking performance. In this paper, we developed a new method based on binary codes to handle the candidate generation step, allowing existing state-of-the-art recommendation modules to be adopted to refine the results. A second contribution was to show that performance can further be improved by adapting these modules to be aware of the generated candidates at training time, using a simple weighted sampling scheme. We showed experiments on several large-scale datasets, where we observed orders-of-magnitude improvements in ranking efficiency, while maintaining or improving upon state-of-the-art accuracy. Ultimately, this means that we have a general-purpose framework that can improve the scalability of existing recommender systems at test time, and surprisingly does *not* require that we trade off accuracy for speed.

Acknowledgements. This work is partly supported by NSF #1750063. We thank Surender Kumar and Lucky Dhakad for their help preparing the Flipkart dataset.

REFERENCES

[1] Nima Asadi and Jimmy J. Lin. 2012. Fast candidate generation for two-phase document ranking: postings list intersection with bloom filters. In *CIKM*.

[2] Fedor Borisyuk, Krishnaram Kenthapadi, David Stein, and Bo Zhao. 2016. CaSMoS: A Framework for Learning Candidate Selection Models over Structured Queries and Documents. In *SIGKDD*.

[3] Zhangjie Cao, Mingsheng Long, Jianmin Wang, and Philip S. Yu. 2017. HashNet: Deep Learning to Hash by Continuation. In *ICCV*.

[4] Evangelia Christakopoulou and George Karypis. 2018. Local Latent Space Models for Top-N Recommendation. In *SIGKDD*.

[5] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *RecSys*.

[6] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*.

[7] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. 2010. The YouTube video recommendation system. In *RecSys*.

[8] Mukund Deshpande and George Karypis. 2004. Item-based top-*N* recommendation algorithms. *ACM TOIS* (2004).

[9] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. 1988. Dynamic Perfect Hashing: Upper and Lower Bounds. In *FOCS*.

[10] 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 *WWW*.

[11] F. Maxwell Harper and Joseph A. Konstan. 2016. The MovieLens Datasets: History and Context. *TiiS* (2016).

[12] Ruining He, Wang-Cheng Kang, and Julian McAuley. 2017. Translation-based Recommendation. In *RecSys*.

[13] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *WWW*.

[14] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. *CoRR* abs/1503.02531 (2015). arXiv:1503.02531 http://arxiv.org/abs/1503.02531

[15] Cheng-Kang Hsieh, Longqi Yang, Yin Cui, Tsung-Yi Lin, Serge J. Belongie, and Deborah Estrin. 2017. Collaborative Metric Learning. In *WWW*.

[16] Binbin Hu, Chuan Shi, Wayne Xin Zhao, and Philip S Yu. 2018. Leveraging meta-path based context for top-n recommendation with a neural co-attention model. In *SIGKDD*.

[17] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *ICDM*.

[18] Qing-Yuan Jiang and Wu-Jun Li. 2018. Asymmetric Deep Supervised Hashing. In *AAAI*.

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

[20] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*.

[21] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and Exact Inner Product Retrieval in Recommender Systems. In *SIGMOD*.

[22] Hao Li, Wei Liu, and Heng Ji. 2014. Two-Stage Hashing for Fast Document Retrieval. In *ACL*.

[23] Wu-Jun Li, Sheng Wang, and Wang-Cheng Kang. 2016. Feature Learning Based Deep Supervised Hashing with Pairwise Labels. In *IJCAI*.

[24] Defu Lian, Rui Liu, Yong Ge, Kai Zheng, Xing Xie, and Longbing Cao. 2017. Discrete Content-aware Matrix Factorization. In *SIGKDD*.

[25] Venice Erin Liong, Jiwen Lu, Yap-Peng Tan, and Jie Zhou. 2017. Deep Video Hashing. *IEEE TMM* (2017).

[26] David C. Liu, Stephanie Rogers, Raymond Shiau, Dmitry Kislyuk, Kevin C. Ma, Zhigang Zhong, Jenny Liu, and Yushi Jing. 2017. Related Pins at Pinterest: The Evolution of a Real-World Recommender System. In *WWW*.

[27] Han Liu, Xiangnan He, Fuli Feng, Liqiang Nie, Rui Liu, and Hanwang Zhang. 2018. Discrete Factorization Machines for Fast Feature-based Recommendation. In *IJCAI*.

[28] Xianglong Liu, Junfeng He, Cheng Deng, and Bo Lang. 2014. Collaborative Hashing. In *CVPR*.

[29] J. J. McAuley, C. Targett, Q. Shi, and A. van den Hengel. 2015. Image-based recommendations on styles and substitutes. In *SIGIR*.

[30] Mohammad Norouzi, Ali Punjani, and David J. Fleet. 2014. Fast Exact Search in Hamming Space With Multi-Index Hashing. *IEEE TPAMI* (2014).

[31] Parikshit Ram and Alexander G. Gray. 2012. Maximum Inner-Product Search using Tree Data-structures. In *SIGKDD*.

[32] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In *UAI*.

[33] Ruslan Salakhutdinov and Geoffrey E. Hinton. 2009. Semantic hashing. *Int. J. Approx. Reasoning* (2009).

[34] Fumin Shen, Chunhua Shen, Wei Liu, and Heng Tao Shen. 2015. Supervised Discrete Hashing. In *CVPR*.

[35] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS). In *NIPS*.

[36] Jiaxi Tang and Ke Wang. 2018. Personalized Top-N Sequential Recommendation via Convolutional Sequence Embedding. In *WSDM*.

[37] Jiaxi Tang and Ke Wang. 2018. Ranking Distillation: Learning Compact Ranking Models With High Performance for Recommender System. In *SIGKDD*.

[38] Yi Tay, Luu Anh Tuan, and Siu Cheung Hui. 2018. Latent Relational Metric Learning via Memory-based Attention for Collaborative Ranking. In *WWW*.

[39] Mengting Wan and Julian McAuley. 2018. Item recommendation on monotonic behavior chains. In *RecSys*.

[40] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2016. Learning to Hash for Indexing Big Data - A Survey. *Proc. IEEE* (2016).

[41] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2018. A Survey on Learning to Hash. *IEEE TPAMI* (2018).

[42] Yair Weiss, Antonio Torralba, and Robert Fergus. 2008. Spectral Hashing. In *NIPS*.

[43] Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA*.

[44] Hanwang Zhang, Fumin Shen, Wei Liu, Xiangnan He, Huanbo Luan, and Tat-Seng Chua. 2016. Discrete Collaborative Filtering. In *SIGIR*.

[45] Longkai Zhang, Houfeng Wang, and Xu Sun. 2014. Coarse-grained Candidate Generation and Fine-grained Re-ranking for Chinese Abbreviation Prediction. In *EMNLP*.

[46] Yan Zhang, Defu Lian, and Guowu Yang. 2017. Discrete Personalized Ranking for Fast Collaborative Filtering from Implicit Feedback. In *AAAI*.

[47] Yan Zhang, Haoyu Wang, Defu Lian, Ivor W. Tsang, Hongzhi Yin, and Guowu Yang. 2018. Discrete Ranking-based Matrix Factorization with Self-Paced Learning. In *SIGKDD*.

[48] Yan Zhang, Hongzhi Yin, Zi Huang, Xingzhong Du, Guowu Yang, and Defu Lian. 2018. Discrete Deep Learning for Fast Content-Aware Recommendation. In *WSDM*.

[49] Han Zhu, Xiang Li, Pengye Zhang, Guozheng Li, Jie He, Han Li, and Kun Gai. 2018. Learning Tree-based Deep Model for Recommender Systems. In *SIGKDD*.

Table 6: Hyper-parameters

	DCF			HashRec			BPR-MF		BPR-MF$^+$		CML		NeuMF	
	r	α	β	r	λ	m	k	λ	k	λ	k	margin	MLP Arch.	k
Ml-20M		0.001	0.001		0.001	16		0.0001		0.01		0.5		
Yelp		0.0001	0.0001		0.1	8		0.0001		0.1		1.0		
Amazon	64	0.001	0.001	64	1.0	4	50	0.01	50	0.1	50	2.0	[200,100,50,25]	25
Goodreads		0.001	0.001		0.01	4		0.0		0.0001		1.0		
Flipkart		-	-		1.0	4		0.001		0.01		2.0		

Appendix

A IMPLEMENTATION DETAILS

We implemented HashRec, BPR-MF, CML, and NeuMF in Tensorflow (version 1.12). For DCF and DPR, we use the MATLAB implementation from the corresponding authors.[9]

For HashRec, BPR-MF, CML, and NeuMF, we use the *Adam* optimizer with learning rate 0.001 and a batch size of 10000. A multi-processing sampler is used for accelerating data sampling. All models are trained for a maximum of 100 epochs. We evaluate the validation performance[10] every 10 epochs, and terminate the training if it doesn't improve after 20 epochs. The bit length r for all hashing-based methods is set to 64, and the embedding size k for ranking models is set to 50.[11]

We use a validation set to search for the best hyper-parameters. The ℓ_2 regularizer λ in HashRec, BPR-MF, and BPR-MF$^+$ is selected from {1, 0.1, 0.01, 0.001, 0.0001, 0}. For DCF, the user and item regularizers α and β are selected from {0.01,0.001,0.0001}, and we set $\alpha = \beta$ for fair comparison as other methods use only one regularizer for both users and items. For CML, the norm of metric embeddings is set to 1 following the paper [15], and the margin in the hinge loss is selected from {0.1, 0.5, 1.0, 2.0}. For NeuMF, we follow the default configurations [13] and a 3-layer pyramid MLP architecture is adopted. As NeuMF has two embeddings for GMF and MLP, we set the embedding size to 25 for each. For DPR, we use the default setting on MovieLens-1M. DPR training on MovieLens-20M did not terminate in 24 hours, hence we only compare against it on MovieLens-1M.

For CIGAR, on all datasets, the number of candidates c=200, the scaling factor α=10/r, sampling ratio h=0.5, and β is increased as shown in Algorithm 1. For multi-index hashing, the number of sub-tables m is set to {16,8,4} depending on dataset sizes.

Table 6 shows the hyper-parameters we used for each model on all datasets.

B EFFICIENCY TEST

We performed the efficiency test (i.e., Section 4.7) on a workstation with a quad-core Intel i7-6700 CPU and a GTX-1080Ti GPU. The GPU is not used except for NeuMF. For MIP Tree [31], we use the implementation from the authors,[12] and the leaf node size is set to 20 following the default setting. For KD-Trees and Ball Trees, we adopt the implementation from the scikit-learn library.[13] A priority queue is employed for choosing the top-k items in linear scan, which costs $O(|\mathcal{I}| \log k)$. The query time of CIGAR consists of obtaining 200 candidates from MIH, performing linear scan on the candidates, and choosing the top-10 items. We assume the queries are independent, and process them sequentially.

C MULTI-INDEX HASHING

We show the procedure of building multi-index hash tables in Algorithm 2. For search, we gradually increase the search radius l until we obtain enough candidates or reach the maximum radius $l_{\max}$. Larger radii may retrieve more accurate neighbors but would cost more time, and we set $l_{\max}$=1 in our experiments. The search procedure is shown in Algorithm 3.

Algorithm 2 Building MIH

```
Input: item binary codes $\mathbf{d}_i \in \{0,1\}^r$ $i \in \mathcal{I}$, the number of substrings $m$
Initialize $m$ hash tables $H_1, H_2, \ldots, H_m$, each containing $2^{r/m}$ buckets
for $i = 1 \rightarrow |\mathcal{I}|$ do
    Split $\mathbf{d}_i$ into $m$ substrings $(s_1, s_2, \ldots, s_m)$
    for $j = 1 \rightarrow m$ do
        Insert item $i$ into the bucket $s_j$ in $H_j$
Output: Hash Tables $H=(H_1, H_2, \ldots, H_m)$
```

Algorithm 3 Querying in MIH

```
Input: Hash Tables $H=(H_1, H_2, \ldots, H_m)$, query codes $\mathbf{b}_u \in \{0,1\}^r$, maximum radius $l_{max}$, number of candidates $c$
$S \leftarrow \emptyset$
Split $\mathbf{b}_u$ into substrings $(s_1, s_2, \ldots, s_m)$
for $l = 0 \rightarrow l_{max}$ do
    for $j = 1 \rightarrow m$ do
        for bucket $b$ with $d_H(b, s_j) = l$ do
            $S \leftarrow S \bigcup$ {items in bucket $b$ of table $H_j$}
    if $|S| \geq c$ then
        break
if $|S| \leq c$ then
    return $S$
else
    sort items in $S$ according to their Hamming distances to $\mathbf{b}_u$, and form a set $S'$ with the $c$ nearest items
    return $S'$
```

[9]Code for DCF is available online: https://github.com/hanwangzhang/Discrete-Collaborative-Filtering

[10]HR@200 for hashing-based methods, and HR@10 for ranking models.

[11]we searched among {10,30,50} and found that 50 works best for all models on all datasets.

[12]https://github.com/gamboviol/miptree

[13]https://scikit-learn.org/stable/modules/neighbors.html