Recommender Forest for Efficient Retrieval

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recommender systems (RS) have to select the top-n items from a massive item set. For the sake of efficient recommendation, RS usually represents users and items as latent embeddings and relies on approximate nearest neighbor search (ANNs) to retrieve the recommendation results. Despite the reduction of running time, the representation learning is independent of ANNs index construction; thus, the two operations can be incompatible, which results in a potential loss of recommendation accuracy. To overcome the above problem, we propose the Recommender Forest (a.k.a., RecForest), which jointly learns latent embedding and index for an efficient and high-fidelity recommendation. RecForest consists of multiple K-ary trees, each of which is a partition of the item set via hierarchical balanced clustering such that each item is uniquely represented by a path from the root to a leaf. Given such a data structure, an encoder-decoder-based routing network is developed: it first encodes user information into user representation; then, leveraging a transformer-based decoder, it identifies the top-n items via beam search. Compared with the existing methods, RecForest brings in the following advantages: 1) the false partition of the near-boundary items can be effectively alleviated by the use of multiple trees; 2) the routing operation becomes much more accurate thanks to the powerful transformer decoder; 3) the branch parameters are shared across different tree levels, making the index to be extremely memory-efficient. The experimental studies are performed on six popular recommendation datasets: with a significantly simplified training cost, RecForest outperforms competitive baseline approaches in terms of both recommendation accuracy and efficiency. The code is available at `https://anonymous.4open.science/r/RecForest-CE76/`.

1 Introduction

Recommender system (RS) is an important way to address the information overload problem. A typical recommender system needs to select the top-n items for users from a massive-scale item set. For the sake of efficient recommendation, RS usually calls for the collaboration of representation learning and Approximate Nearest Neighbour search (ANNs). In the first place, users and items are represented by embeddings in the same latent space; in the second place, the item embeddings are organized with a specific ANNs index, like SCANN and HNSW, such that the top-n recommendation to the user can be efficiently accomplished. Despite that the recommendation process is greatly accelerated from the above workflow, the recommendation quality will probably be restricted, given that the representation model is independently learned and can be incompatible with the ANNs index.

In recent years, many efforts have been devoted to alleviating the incompatibility between the representation model and the ANNs index, especially the effort on joint optimization of both components. One representative class of works is the tree-based deep model (TDM) [23] and the joint optimized tree-based model (JTM) [22] from Alibaba. In both works, the item set is organized with a binary tree structure: each internal node acts like a cluster center, and each leaf node corresponds to a

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

unique item. On top of the tree structure, a preference model is learned to route from the root to the leaf nodes for the top-n recommendation results. These works achieve empirical gains over the conventional two-stage methods; besides, they preserve competitive retrieval efficiency as the time cost is logarithmic to the size of the item set.

However, the existing tree-based recommenders are still restricted in many aspects. Firstly, the item set is hierarchically partitioned; as a result, it is challenging to route to items located around partition boundaries. Secondly, a routing decision is made without consideration of the routing trajectory (i.e., from the root to the direct ancestor of the current node), so the accuracy of beam search can be limited. Thirdly, the tree-based index can be memory-consuming, given that the number of internal nodes is at the same magnitude as the leaf nodes (i.e., the number of the items). Last but not least, the existing methods call for joint adaptation of the representation model and tree index, so that the tree structure needs to be repetitively updated, resulting in a significant cost for the training stage.

To overcome the limitations, we propose a novel framework, Recommender Forest (a.k.a., RecForest) for an efficient and high-fidelity recommendation. RecForest is highlighted for the following features.

- RecForest consists of multiple K-ary trees, each of which is a partition of the item set based on balanced hierarchical clustering. With the construction, the retrieval of the near-boundary items can be effectively improved, as evidenced in Section 3.6.1, since a near-boundary item missed in one tree can be retrieved back on another tree.
- RecForest leverages a transformer decoder for beam search. On top of such a decoder, the routing trajectory, i.e., from the root node to the current node, can be jointly considered when the next routing decision is made. Compared with the previous methods which merely take account of the current node, the beam search becomes much more accurate since the routing trajectory is fully utilized, as analyzed in Section 3.4.
- The tree parameters are shared across different tree levels; in other words, there are mere K vectors (corresponding to the K different branches) in each K-ary tree. Thanks to parameter sharing, RecForest becomes much more memory-efficient compared with the existing tree-based recommenders, as demonstrated in Table 1 theoretically and in Table 3 empirically.
- Given the above settings, RecForest becomes much less sensitive to the partition of the item set. As a result, without any tree update, RecForest can perform remarkably better than TDM and JTM with repetitive tree update, as shown in Table 3. Therefore, RecForest can avoid the repetitive adaption of the tree structure, saving a considerable portion of the training cost.

We perform comprehensive evaluations on six popular recommendation datasets. According to our experiment results, RecForest notably outperforms the existing tree-based recommenders in terms of recommendation quality and efficiency. Besides, we empirically verify that RecForest can be effectively trained with much less time cost, indicating its strong usability in real-world scenarios.

2 Recommender Forest

2.1 Preliminary

As mentioned above, RS usually represents user and item as latent embeddings, where the user's preferences for the items are measured by the inner product of the embeddings [13, 1, 9]. As a result, the top-n recommendation is boiled down to the maximum inner product search (MIPS) problem. For the sake of efficient recommendation, various ANNs indexes are leveraged in practice, such as the tree-based index [12, 8]; the hash-based index [15, 16, 5, 11], the quantization-based index [2, 19, 3] and the graph-based index [10, 17, 21], etc. These indexes have been well implemented by toolkits, like FAISS, which greatly facilitates the deployment of recommendation systems in practice. However, one limitation of the current ANN-based methods is that the index construction and the representation learning are decoupled, which will probably introduce incompatibility between both modules.

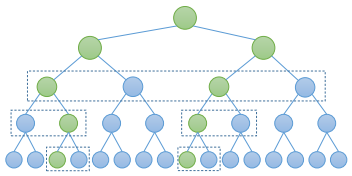
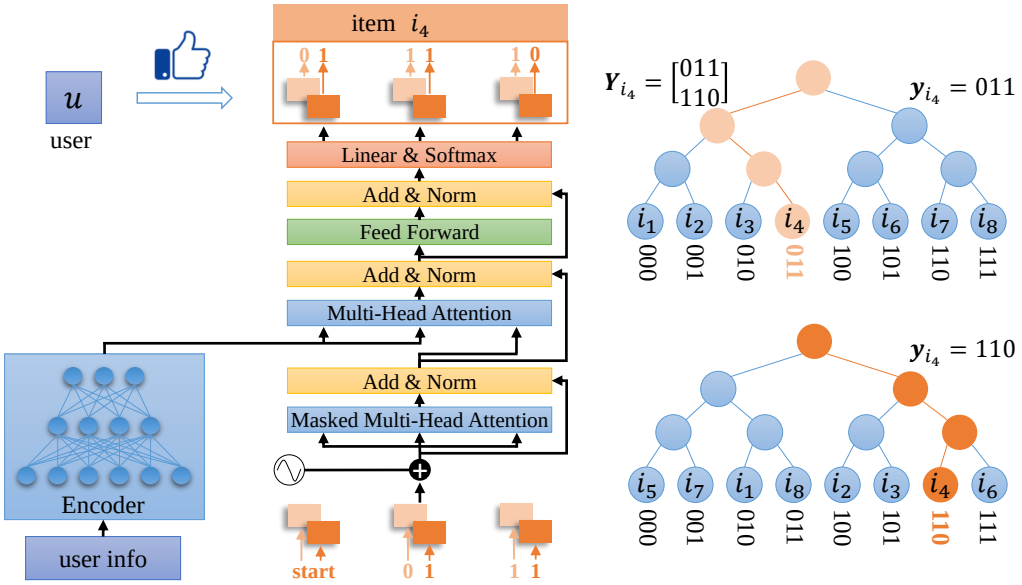


Figure 1: Beam search over a binary tree. The beam size is set 2. The nodes in the dotted box of each layer are candidates for scoring, where green nodes are selected.



(a) Encoder-Decoder Framework of RecForest (b) Forest of 2 Balanced Binary Trees

Figure 2: Illustration of RecForest framework.

To mitigate the above problem, recent works were proposed to jointly optimize the representation model and index, where two representative works are TDM [23] and JTM [22]. Both methods are based on the tree structures, where the top-n recommendation can be efficiently accomplished by a layer-wise beam search. For example, as shown in Figure 1, when the beam size is 2, these models will score the four children of previously retrieved 2 nodes, from which the two largest children will be selected. Such a process is iteratively performed until the final top-2 leaf nodes are retrieved.

2.2 Overview

The framework of RecForest is overviewed as follows. First of all, the item set is partitioned based on hierarchical balanced Kmeans, so that a tree structure is generated. The tree structure will enable an item to be efficiently retrieved in $O(\log N)$ time complexity (N is the number of the items). Knowing that the items located close to the partition boundaries may get falsely assigned to a branch missing from the retrieval process, we propose to leverage multiple diversified trees, where the missing probability can be largely reduced. One item is corresponding to a leaf node in the tree, which can be represented by a path stretching from the route. Suppose that the K-ary tree is utilized, each item can be represented by a sequence of branch ID $(0, ..., K-1)$, whose length is $\lceil \log_K N \rceil$. For example, in Figure 2(b), the item i_4 is represented by "011" and "110" in the corresponding binary trees, respectively. Given the hierarchical numbering of the items, the recommendation turns out to be a sequence-to-sequence problem: based on the encoded user representation, paths to the most preferable items are progressively decoded via beam search, from which the top-n recommendation is made. The overall framework of RecForest is shown in Figure 2(a).

For ease of understanding, let's first summarize the notations used in this paper. Let $\boldsymbol{x}_u$ denote the user information of u, which can be either a behavior sequence or a user feature vector. Let $\boldsymbol{y}_i \in \{0, K-1\}^H$ be the item i's routing sequence of length H and $\boldsymbol{Y}_i \in \{0, K-1\}^{T \times H}$ be a concatenation of item i's routing sequences of T trees. Formally, RecForst models the conditional probability $P(\boldsymbol{Y}_i | \boldsymbol{x}_u)$ with the seq2seq architecture.

2.3 Construction of Tree and Forest

As mentioned, the item set is organized by a tree structure, where semantically close items should fall into the same branch as much as possible. We will construct multiple trees so as to alleviate the false assignment of items around partition boundaries. Detailed workflows are illustrated as follows.

2.3.1 Tree Construction

The tree is expected to be balanced for the sake of efficient beam search on the tree. Thus, we leverage hierarchical balanced clustering, where each cluster is evenly divided into K child clusters until each child cluster merely contains one single item. We propose to randomly sample $K^H - N$ items ($H = \lceil \log_K N \rceil$ is the height of the tree) from the entire item set, such that the constructed tree will be a complete K-ary tree.

We suggest the following two methods for the balanced partition of the item set. **Random**: For each cluster, the included items are randomly partitioned into K equally-sized subsets. Such a method is extremely simple; yet, the semantic relationship between items is ignored. **Balanced Kmeans** (short for Kmeans): Particularly, we first pre-train item embeddings based on an arbitrary off-the-shelf recommendation model. In our experiment, we use the Deep Interest Network (DIN) [20] for sequential recommendation and the Bayesian Personalized Ranking (BPR) [13] for non-sequential recommendation, given their effectiveness and popularity in corresponding scenarios. With such a recommendation model, the items' semantic closeness can be measured by their embedding similarity, as relevant items will be close to each other in the latent space. Then, for each cluster, the included items are evenly partitioned via Kmeans w.r.t. their embedding similarity. Due to the space limitation, details about balanced Kmeans clustering are given in the appendix.

2.3.2 Forest Construction

Different trees are desirable of being diversified such that the boundary items can be better covered from beam search. To this end, we propose a simple but effective approach, which combines the random and Kmeans tree construction. Particularly, for each cluster with n items, $n \mod K$ items are first randomly chosen, from which one item is sampled with replacement into each child cluster; then, the remaining items are further partitioned into the K equally-sized groups via balanced Kmeans. By doing so, different trees are naturally diversified due to the inherent randomness.

2.4 RecForest Enocoder

As shown in Figure 2(a), the encoder is utilized to encode user information and output user representation. Any neural network based encoders are compatible with our framework. We mainly consider the sequential scenario and non-sequential scenario, where a sequence encoder and a feature encoder is used to encode user behavior sequences and feature vectors, respectively.

Feature Encoder In the non-sequential scenarios, user information is denoted by a feature vector, i.e. $\boldsymbol{x}_u$. We simply use a MLP for the feature encoder. $\boldsymbol{z}_u = \text{MLP}(\boldsymbol{x}_u)$. When only user id is available, we can take its embedding as user representation, without any non-linear transformation.

Sequence Encoder In the sequential scenarios, user information is an item sequence. Concretely, $\boldsymbol{x}_u = [i_{t_1}, i_{t_2}, \ldots, i_{t_m}]$, where the item $i_{t_j} (1 \leq j \leq m)$ is the j-th interacted item of user u. It is straightforward to take item features into account. In view of Transformer's [18] state-of-the-art performance in seq2seq tasks like machine translation and speech recognition, we utilize the Transformer encoder for the sequence encoder. In particular, $\boldsymbol{z}_u =$ transformer-encoder$(\boldsymbol{x}_u)$, where the length of $\boldsymbol{z}_u$ is the same as $\boldsymbol{x}_u$. Due to space limitation, more details about the Transformer encoder are provided in the appendix.

2.5 RecForest Decoder

The decoder is to predict the next branch (routing decision) given the routing trajectory and the encoder's outputs of user representation. As shown in Figure 2(a), we apply the Transformer [18] decoder for this task due to its powerful capability. From down to up, following the branch embedding and positional embedding, the Masked Multi-Head Attention [18] is utilized to model the complex dependence among branches at different layers. Taking the outputs from the Masked Multi-Head Attention, Multi-Head Attention [18] is followed to encode the complex interactions between user representation and routing trajectory representations. The FFN layer then enhances non-linearity of context-aware routing trajectory representations. A linear layer with the softmax activation is applied for predicting the next routing decision. Note that this decoder can also handle the non-sequence scenarios since a user feature vector can be considered as a sequence only with one element.

2.6 Training

As depicted in Figure 2(b), the routing trajectory of item i_4 is [0,1,1] and [1,1,0] on the two trees, respectively. The framework uses [start,0,1] to predict [0,1,1] and uses [start,1,1] to predict [1,1,0]. 'start' is a special symbol for initializing the sequence, which actually corresponds to the tree root. According to experimental results, we do not share the transformer decoder among trees, i.e., $P(\boldsymbol{Y}_i|\boldsymbol{x}_u) = \prod_k P_{\theta_k}(\boldsymbol{Y}_i[k]|\boldsymbol{x}_u)$, where θ_k indicate the parameters of the k-th tree. In this case, all trees share the same training procedure, so we illustrate the training with one tree. At each layer, the training task is to predict the next routing decision given routing trajectory, which corresponds to a multi-class classification problem. Given a user u and its interacted item i, the loss for optimizing encoder and decoder is formulated as follows:

$$\mathcal{L}(u,i) = -\sum_{h=0}^{H-1} \log \text{Prob}(y_i^h|\text{cat}(\text{start}, \boldsymbol{y}_i^{[0:h]}), \boldsymbol{z}_u) \tag{1}$$

where y_i^h and $\boldsymbol{y}_i^{[0:h]}$ indicate the branch (routing decision) at the h-th layer and routing trajectory until the h-th layer respectively, and 'cat' means concatenation.

2.7 Inference

Since each leaf in the tree corresponds to an item, RecForest transforms top-n recommendation into the sequence prediction problem. In particular, we first apply the encoder for deriving user representation and leverage the decoder to generate the top-n routing sequences, i.e., paths from the root node to leaf nodes on each tree, based on beam search. Note that each item can correspond to multiple leaf nodes (i.e. routing sequences) in each tree since some items are randomly sampled for building a complete K-ary tree. Therefore, the generated items should be duplicated. Let C denote the union of generated candidate items from the forest. For $c \in C$, we first compute the logarithmic probability $\sum_{h=0}^{H-1} \log \text{Prob}(y_i^h|\text{cat}(\text{start}, \boldsymbol{y}_i^{[0:h]}), \boldsymbol{z}_u)$ in each tree and then sum logarithmic probabilities over all trees as the score of the item c. We then select the top-n items from C for recommendation.

2.8 Complexity Analysis

Notation Denote by D item embedding size, B the beam search size, K the number of branches, and I the training set size. Denote by T the number of trees in RecForest and the times of updating the tree in JTM and TDM. In SCANN, denote by K_{vq} the number of centroids in VQ, K_{pq} the number of centroids in PQ, M the number of subspaces and W the number of probed VQ cells.

Time Complexity In inference, the beam search can be finished within $O(TKB\log_K N)$ since the probabilities of KB branches at each level of all trees need to be calculated. Hierarchical Kmeans clustering can be done within $O(NK\log_K N)$ to build a tree index so that it takes $O(TNK\log_K N)$ to build the forest index.

Space Complexity The memory consumption of the recommendation model is inevitable in any recommender systems, so we mainly focus on the size of the index structure. In RecForest, we only need to keep the tree structure and T matrices with size $K \times D$, denoting branch embeddings of T trees. In TDM and JTM, beside the tree structure, they also store all tree node embeddings, where the number of tree nodes is at the same magnitude as the number of the items.

Table 1: Complexity analysis. The SCANN indexing time only considers the encoding period given codebooks for fair comparison. The time complexity of IPNSW directly is taken from the original paper of HNSW, which is derived with exact Delaunay graphs.

Complexity	RecForest	JTM	TDM	IPNSW	SCANN
Inference Time	$O(TKB\log_K N)$	$O(B\log N)$	$O(B\log N)$	$O(\log N)$	$O(K_{vq} + K_{pq} + \frac{W}{K_{vq}} \cdot N)$
Indexing Time	$O(TNK\log_K N)$	$O(TI\log N)$	$O(TN\log N)$	$O(N\log N)$	$O(NMK_{pq})$
Index Size	$O(TKD)$	$O(ND)$	$O(ND)$	$O(ND)$	$O(K_{vq}D + K_{pq}D)$

We summarize complexity analysis of typical algorithms in Table 1, which shows RecForest enjoys a small index memory cost and low inference time. This is because a smaller beamsize and fewer beam search are only required, and tree node representations are not stored but computed on the fly.

3 Experiments

We conduct the experiments to answer the following research questions: **RQ-1**: *Does RecForest outperform the SOTA efficient recommenders in the tradeoff between efficiency and accuracy?* **RQ-2**: *Does the forest-based index improve near-boundary item retrieval?* **RQ-3**: *How much effect positional embedding can take?* The experiments are carried out in both sequential scenarios and non-sequential scenarios, where the sequence encoder and feature encoder is utilized, respectively. Due to space limitations, we only report the results in the sequential scenario. Other results are provided in the appendix. These experiments are done on a Linux server with Tesla V100 GPUs.

3.1 Dataset

We evaluate the RecForest with six real-world recommendation datasets, which can be downloaded from the url[1]. The datasets are **MovieLens 10M** (abbreviated as **Movie**), **Amazon Books** (abbreviated as **Amazon**), **Tmall Click** (abbreviated as **Tmall**), **Gowalla Check-in Dataset** (abbreviated as **Gowalla**), Microsoft News Dataset (abbreviated as **MIND**). Since some datasets only include rating-based explicit feedback, they should be converted into implicit feedback for RecForest's inputs. These datasets are pre-processed by filtering the users who interact with no more than 15 items. The overall information of datasets is summarized in Table 2.

Table 2: Statistics of Datasets

Dataset	#User	#Item	#Interaction	Density
Movie	69,878	10,677	10,000,054	1.34%
Amazon	29,980	67,402	2,218,926	0.11%
Tmall	139,234	135,293	10,487,585	0.05%
Gowalla	13,583	71,436	977,425	0.10%
MIND	36,281	7,129	5,610,960	2.16%
Yelp	26,031	35,294	1,713,759	0.19%

3.2 Baselines

In the part, we report the results of the sequential recommendation scenario, so we compare the proposed RecForest with two-stage indexes, IPNSW [10] and SCANN [3], with learnable indexes, TDM [23] and JTM [22] as well as with YoutubeDNN [1] and DIN [20] based on the brute-force retrieval. Note that brute-force-based YoutubeDNN and DIN are time-prohibitive in online services, but they can be considered as two strong baselines. IPNSW and SCANN are built over item embedding of a recommender. The recommender shares the same encoder as RecForest, but replaces the decoder with flat item embedding. It is then trained with the same loss as DIN. Both TDM and JTM make a routing decision simply based on current node without the consideration of routing trajectories. More details about baselines can be referred to in the appendix.

3.3 Experimental Settings

In each dataset, we randomly choose 10% users as validation users, 10% users as test users, and all the left users as training users. Following TDM and JTM, we use a slide window to split user-item interaction histories into slices of length 70 at most. For training users' data, the first 69 interactions are used for input context and the 70-th item is regarded as the ground truth of prediction. For data of both validation users and test users, we regard the first half as context and others as ground truth.

The latent dimensionality is set to 96 in all methods. Here, for a fair comparison of running time, we implement SCANN and IPNSW with PyTorch. To construct the quantization-based index, we follow SCANN's default settings. To construct the graph index of IPNSW, the maximum degree of each node is set to 16. The beam size for any beam search is set to 100 unless specified. To learn models, the learning rate is all set to 1e-3 with exponential decay. The items' representations for constructing trees are obtained from item embedding of the pre-trained DIN on each dataset.

3.4 Comparison with Baselines

Settings We compare RecForest with the baselines on all datasets. RecForest uses 2 trees on the Movie and MIND, and 4 trees on other datasets. Other settings can be referred to in Section 3.3.

[1] https://drive.google.com/drive/folders/1ahiLmzU7cGRPXf5qGMqtAChte2eYp9gI

Table 3: Comparison with Baselines w.r.t NDCG@20 and NDCG@40, index memory cost (MB), and inference time (second). The bold fonts indicate the best performance.

	NDCG@20	NDCG@40	Memory	Time	NDCG@20	NDCG@40	Memory	Time
Method	**Movie**				**Amazon**			
DIN	**0.5440**	0.5473	-	193.87	**0.2766**	**0.3039**	-	492.64
YoutubeDNN	0.5329	**0.5484**	-	**29.38**	0.2195	0.2491	-	**120.91**
JTM	0.5149	0.5075	10.80	12.05	0.1533	0.1683	75.99	6.64
TDM	0.4684	0.4651	10.80	9.33	0.0856	0.0949	75.99	6.61
SCANN	0.4665	0.4695	3.64	18.64	0.1529	0.1780	14.66	4.48
IPNSW	0.5330	0.5486	10.08	15.52	0.2255	0.2548	66.46	10.28
RecForest	**0.5580**	**0.5682**	**3.21**	**8.33**	**0.2339**	**0.2576**	**7.32**	**3.79**
Method	**Gowalla**				**Tmall**			
DIN	**0.2798**	**0.3095**	-	186.41	**0.2275**	**0.2491**	-	4057.69
YoutubeDNN	0.2312	0.2637	-	**53.55**	0.1736	0.1975	-	**1086.75**
JTM	0.2595	0.2484	77.56	2.64	0.0749	0.0849	151.19	30.11
TDM	0.1723	0.1775	77.56	2.55	0.0257	0.0272	151.19	29.42
SCANN	0.1839	0.2083	15.48	1.86	0.1105	0.1226	28.10	20.88
IPNSW	0.2464	0.2805	70.39	4.73	0.1696	0.1902	132.72	52.90
RecForest	**0.3783**	**0.3963**	**7.39**	**1.82**	**0.2059**	**0.2261**	**9.29**	**18.88**
Method	**MIND**				**Yelp**			
DIN	**0.7399**	**0.7399**	-	62.98	**0.2825**	**0.3117**	-	170.25
YoutubeDNN	0.7349	0.7336	-	**52.14**	0.2518	0.2850	-	**48.51**
JTM	0.5956	0.5505	6.62	5.48	0.1014	0.1300	39.47	4.21
TDM	0.5615	0.5198	6.62	5.51	0.1547	0.1515	39.47	4.34
SCANN	0.5987	0.5713	3.20	19.51	0.1729	0.2012	8.44	3.58
IPNSW	0.7346	0.7331	6.95	8.99	0.2562	0.2906	34.74	8.92
RecForest	**0.7583**	**0.7579**	**3.18**	**4.61**	**0.2766**	**0.3031**	**6.81**	**3.57**

Results All the results are shown in Table 3, where the index memory cost of RecForest indicates the decoder's memory consumption. The following observations answers the **RQ-1**.

- **RecForest significantly outperforms all efficient recommenders with indexes on all datasets w.r.t NDCG@20 and NDCG@40**. The improvements over the best baselines w.r.t NDCG@20 are 4.69%, 3.72%, 45.78%, 21.40%, 3.23%, 7.96% on the Movie, Amazon, Gowalla, Tmall, MIND, and Yelp dataset, respectively. Its superiority over two-stage baselines (i.e., IPNSW and SCANN) demonstrates the effectiveness of joint representation learning, although the graph-based index shows a strong performance. The higher accuracy than TDM and JTM indicates the benefit from the use of routing trajectory, multiple trees and the powerful transformer decoder.

- **RecForest performs better than brute-force-based YoutubeDNN on all datasets and even than brute-force-based DIN on the Movie, Gowalla, and MIND datasets.** Note that we do not apply DIN for post-reranking, otherwise the retrieval accuracy can be further improved. This again confirms the superpower of the transformer decoder. Note that after fine-tuning the hyper-parameters, YoutubeDNN performs well compared to TDM and JTM, which may not be consistent with the results in TDM and JTM.

- **RecForest also enjoys the smallest index memory cost compared to other methods**, which is now supported empirically and theoretically. This is because RecForest only embeds the routing decision and compute tree node representation on the fly, compared to directly embedding all tree nodes in TDM and JTM. RecForest even costs fewer memories for index than the most lightweight PQ-based index – SCANN, which only stores the codes of data points and codebooks. In spite of good retrieval accuracy, the graph-based index IPNSW is memory-heavy due to the storage of graphs and data points. Note that both DIN and YoutubeDNN are based on exhaustive search so they don't consume memory for indexes.

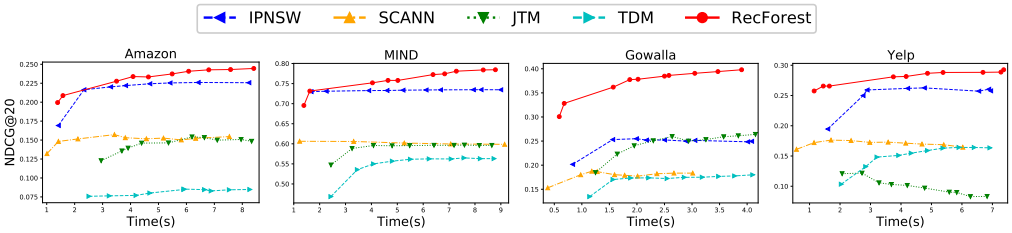


Figure 3: Tradeoff between Efficiency and Accuracy

- **RecForest is almost as fast as and sometimes even faster than SCANN for the top-n item retrieval**, which is considered as the SOTA index for MIPS in retrieval efficiency and accuracy. This benefits from transforming the top-n recommendation into sequence prediction, which can be efficiently accomplished by beam search on GPU. Note that the running time cost of IPNSW seems inconsistent with Table 1, this is because only the approximate proximity graphs instead of exact Delaunay graphs can be efficiently constructed for query in practice. Theoretical results show that the RecForest is almost as efficient as TDM and JTM, while the experimental results indicate RecForest is slightly faster. On the one hand, this is because we use the comparatively large branches (much larger than 2 used in TDM and JTM), remarkably decreasing the tree depth and thus reducing the times of beam search. On the other hand, because RecForest uses the simple function for routing decision and enables high parallelism due to the use of transformer decoder.

3.5 Extensive Study between Efficiency and Accuracy

Settings The efficiency-accuracy curve is a commonly-used standard for evaluating the ANNs index, so we also provide this result for better illustrating the superiority of RecForest. The study is mainly investigated on the Amazon, MIND, Gowalla, and Yelp dataset, since the other two datasets show a similar trend. To vary the retrieval time of RecForest, we adjust the beam size of beam search from 10 to 100 with step 10, where the forest consists of at most 5 trees. For TDM, JTM, and IPNSW, we adjust the beam size from 10 to 200 with step 10. For SCANN, we adjust the number of probe cells from 50 to 2,000 with step 100.

Results The curves of different algorithms between NDCG@20 and query time are shown in Figure 3. The following observations confirms the answer of **RQ-1**. First, **RecForest strikes the best balance between query time and retrieval accuracy on all the four datasets**, since the curve stands on top of the others. This confirms the superiority of RecForest to these competing baselines. The advantage is the most significant on the Gowalla dataset, evidenced by the biggest gap between RecForest and IPNSW. Second, **With the increase of beam size, the accuracy of RecForest can improve more significantly than baselines.** This benefits from the powerful transformer decoder and the novel training paradigm as well as the forest-based index, such that the representation model and tree indexes are jointly learned better.

3.6 Ablation Study

3.6.1 Effect of Forest Construction

Settings We investigate three aforementioned ways (in Section 2.3.2) of constructing a forest with at most 10 trees, where the branch number of each tree is set to 4.

Results The results on the Amazon, Gowalla, MIND, and Yelp datasets are reported in Figure 4. We can observe that **Kmeans with randomness (i.e. Random+Kmeans) always performs best and improves with the increasing number of trees while the Random performs worst and does not improve as much as Random+Kmeans when the tree number increases.** The reason why the Random does not work well is that the semantic requirement is not satisfied. Since Kmeans utilizes the semantic information, it performs better than the Random, but is not affected a lot by growing the number of trees. This can be explained by the lack of diversity between trees. This answers the **RQ-2**: since the use of forest-based index can lead to significant improvement of retrieval accuracy, particularly when tree construction takes semantics into account, *the forest-based index improves near-boundary item retrieval*.

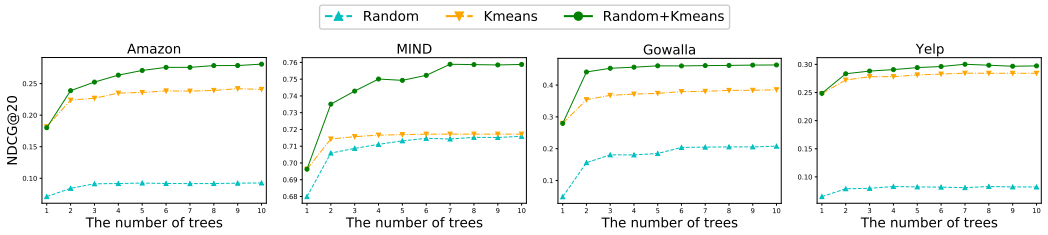


Figure 4: Ablation study: Different construction ways for forests.

3.6.2 Effect of Branch Number K

Settings We conduct evaluations on a single Kmeans-constructed tree on the Amazon, MIND, Gowalla and Yelp dataset, by varying the branch number in {2,4,8,10,16,18}.

Results The results are shown in Table 4. We can observe that **the retrieval accuracy of RecForest improves with the growing number of branches, particularly from 2 branches to 4.** The improvement on the Gowalla and Yelp datasets is the most significant. This observation can be explained as follows: First, the vocabulary size grows, so the decoder can be supervised by more challenging discriminative signals and the decoders' parameters grow. Secondly, the routing sequences would be shorted, reducing the error accumulation of sequence prediction to some extent.

Table 4: Effect of branch number

#Branch	NDCG@20			
	Amazon	MIND	Gowalla	Yelp
2	0.1624	0.6900	0.2353	0.1951
4	0.1912	0.6935	0.2795	0.2269
8	0.1864	0.6966	0.2938	0.2483
10	0.2022	0.6954	0.2843	0.2577
16	0.2031	0.6977	0.2957	0.2437
18	**0.2088**	**0.7154**	**0.3337**	**0.2581**

3.6.3 Effect of Positional Embedding

Settings Since the routing sequence is used for representing an item, the order of routing decisions is important. To this end, we compare four positional embeddings: (1) **None** indicates that there is no positional embedding; (2) Absolute positional embedding (**Abs** for short) encodes the absolute positions which range from 1 to maximum length of sequence by learnable parameters; (3) Relative Key [14] (**RelK** for short) focuses on attention and relative distance between decisions; (4) Relative Key Query [6] (**RelKQ** for short) is a refined Relative Key. Please refer to [6] for more details.

Results The results based on from single Kmeans-constructed tree on the Amazon, MIND, Gowalla, and Yelp datasets are summarized in Table 5. We can observe that **positional embedding takes a remarkable effect on improving the retrieval capacity while different positional embedding methods do not have a significant difference in retrieval accuracy**, which answers the **RQ-3**. The effects of positional embedding are more significant on the Gowalla and Yelp datasets.

Table 5: Effect of positional embedding

Pos. Emb.	NDCG@20			
	Amazon	MIND	Gowalla	Yelp
None	0.1704	0.6476	0.1697	0.1988
Abs	0.1805	**0.6986**	**0.2632**	**0.2505**
RelK	0.1824	0.6942	0.2617	0.2402
RelKQ	**0.1840**	0.6917	0.2588	0.2492

4 Conclusion and Future Work

In this paper, we propose the Recommender Forest for efficient recommendation, which can be simply trained within the sequence-to-sequence framework. RecForest enjoys a small index memory cost, low inference time, and highly-accurate recommendation even without updating the tree structure. The extensive study on six real-world recommendation datasets shows that RecForest becomes a state-of-the-art efficient recommender. In the future, we explore multi-task learning, non-autoregressive prediction, and index structure learning as well as a more general framework.

References

[1] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016.

[2] Ruiqi Guo, Sanjiv Kumar, Krzysztof Choromanski, and David Simcha. Quantization based fast inner product search. In *Artificial Intelligence and Statistics*, pages 482–490. PMLR, 2016.

[3] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pages 3887–3896. PMLR, 2020.

[4] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

[5] Qiang Huang, Guihong Ma, Jianlin Feng, Qiong Fang, and Anthony KH Tung. Accurate and fast asymmetric locality-sensitive hashing scheme for maximum inner product search. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1561–1570, 2018.

[6] Zhiheng Huang, Davis Liang, Peng Xu, and Bing Xiang. Improve transformer models with better relative position embeddings. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 3327–3335. Association for Computational Linguistics, 2020.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[8] Noam Koenigstein, Parikshit Ram, and Yuval Shavitt. Efficient retrieval of recommendations in a matrix factorization framework. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 535–544, 2012.

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

[10] Stanislav Morozov and Artem Babenko. Non-metric similarity graphs for maximum inner product search. *Advances in Neural Information Processing Systems*, 31, 2018.

[11] Behnam Neyshabur and Nathan Srebro. On symmetric and asymmetric lshs for inner product search. In *International Conference on Machine Learning*, pages 1926–1934. PMLR, 2015.

[12] Parikshit Ram and Alexander G Gray. Maximum inner-product search using cone trees. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 931–939. ACM, 2012.

[13] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence, UAI 2009*, 05 2012.

[14] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 464–468. Association for Computational Linguistics, 2018.

[15] Anshumali Shrivastava and Ping Li. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[16] Anshumali Shrivastava and Ping Li. Improved asymmetric locality sensitive hashing (alsh) for maximum inner product search (mips). In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, UAI'15, page 812–821, Arlington, Virginia, USA, 2015. AUAI Press.

[17] Shulong Tan, Zhixin Zhou, Zhaozhuo Xu, and Ping Li. On efficient retrieval of top similarity vectors. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5239–5249, 2019.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[19] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Dan Holtmann-Rice, David Simcha, and Felix X Yu. Multiscale quantization for fast similarity search. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 5749–5757, 2017.

[20] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1059–1068, 2018.

[21] Zhixin Zhou, Shulong Tan, Zhaozhuo Xu, and Ping Li. Möbius transformation for fast inner product search on graph. In *Advances in Neural Information Processing Systems*, pages 8218–8229, 2019.

[22] Han Zhu, Daqing Chang, Ziru Xu, Pengye Zhang, Xiang Li, Jie He, Han Li, Jian Xu, and Kun Gai. Joint optimization of tree-based index and deep model for recommender systems. In *NeurIPS*, 2019.

[23] Han Zhu, Xiang Li, Pengye Zhang, Guozheng Li, Jie He, Han Li, and Kun Gai. Learning tree-based deep model for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 1079–1088, New York, NY, USA, 2018. Association for Computing Machinery.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section **??**.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] Please see the future work in section 4.
   (c) Did you discuss any potential negative societal impacts of your work? [No] For the proposed RecForest model, we have not seen negative societal impacts
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We provide the URL in the abstract.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Please see section 3.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendices for the Submission: Recommender Forest for Efficient Retrieval

Our appendices include the following content:

- Sec.A: the details about forest construction.
- Sec.B: the illustration of Transformer encoder.
- Sec.C: the pseudocode of training and inference.
- Sec.D: the metrics used in performance evaluation.
- Sec.E: the experiment details in sequential recommendation scenario.
- Sec.F: the experiments conducted in non-sequential scenarios.
- Sec.G: the additional ablation studies.

A Forest Construction

A.1 Balanced Kmeans

For the sake of retrieval efficiency, the constructed tree is expected to be a complete K-ary tree. However, the item number N does not necessarily meet this requirement. We grow the dataset by inserting $K^H - N$ randomly sampled items from itself, such that it exactly contains K^H items, where $H = \lceil \log_K N \rceil$ is the tree height. We use the Balanced Kmeans in Algorithm 1 for hierarchical itemset partition. In particular, supposing a cluster contains $K \times m$ items, the algorithm splits them into K clusters such that each cluster exactly contains m items. At each iteration, we first assign each centroid the m closest unassigned items in order and then update the centroids according to item assignment. The clustering algorithm terminates until the assignments do not change any more.

Algorithm 1: Balanced-Kmeans

Data: Item set $\mathbb{D}$, cluster number K
Result: $\{\mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_K\}$

$m \leftarrow \frac{|\mathbb{D}|}{K}$;
Randomly initialize K centroids $\mathbb{C} = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_K\}$;
repeat
  $\mathbb{U} \leftarrow \mathbb{D}$;
  for $k = 1 \rightarrow K$ **do**
    $U_s \leftarrow$Sort items in $\mathbb{U}$ ascendingly according to distance from $\boldsymbol{c}_k$;
    $\mathbb{D}_k \leftarrow U_s[0 : m-1]$;
    $\boldsymbol{c}_k \leftarrow \frac{1}{m} \sum_{\boldsymbol{x} \in \mathbb{D}_k} \boldsymbol{x}$;
    $\mathbb{U} \leftarrow \mathbb{U} \setminus \mathbb{D}_k$;
until *Assignments do not change*;

A.2 Tree Construction

As described in Section 2.3.2, each tree is constructed by combining **Random** partition and Balanced **Kmeans** to guarantee both diversity and semantic closeness of similar items. Regarding the **Random** strategy, for each cluster, the included items are randomly partitioned into K equally-sized subsets. When it is combined with balanced Kmeans, for each cluster with n items, $n \mod K$ items are first randomly picked, from which one item is sampled with replacement into each child cluster, and the remaining items are partitioned based on the Balanced Kmeans. This method is shorten for **Random+Kmeans**. By applying the **Random+Kmeans** strategy recursively until each cluster only contains one item, the tree index can be constructed successfully. Repeating the tree construction can generate diverse trees to form the forest. We summarize the procedure in Algorithm 2.

B Transformer encoder

The Transformer encoder, shown in Figure 5, is to capture user's interests from the sequential interactions $\boldsymbol{x}_u = [i_{t_1}, i_{t_2}, \ldots, i_{t_m}]$. From down to up, following the interaction embedding and positional

Algorithm 2: Tree construction

Data: Dataset $\mathbb{D}$, branch number K, random ratio δ
Result: Constructed tree

$Q \leftarrow$ A queue with an element $\mathbb{D}$;
while Q *is not empty* **do**
  $\mathbb{D}_{top} \leftarrow$ pop(Q);
  $n \leftarrow |\mathbb{D}_{top}|$;
  $\mathbb{D}^0 \leftarrow$ Randomly draw $n \mod K$ items from $\mathbb{D}_{top}$ without replacement ;
  $\mathbb{D}_{top} \leftarrow \mathbb{D}_{top} \setminus \mathbb{D}^0$;
  $\{\mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_K\} \leftarrow$Balanced-Kmeans($\mathbb{D}_{top}, K$);
  for $k = 1 \rightarrow K$ **do**
    $i \leftarrow$ Sample one item with replacement from $\mathbb{D}^0$;
    push(Q, $\{i\} \cup \mathbb{D}_k$);

embedding, Multi-Head Attention [18] is utilized to model the interdependence of user's each behavior. Taking the output of Multi-Head Attention, the FFN layer is to enhance the non-linearity of user's behavior representations. Finally, the encoder output, denoted as $\boldsymbol{z}_u = [\boldsymbol{z}_u^1; \boldsymbol{z}_u^2; \ldots; \boldsymbol{z}_u^m]$, is utilized to characterize user representation.

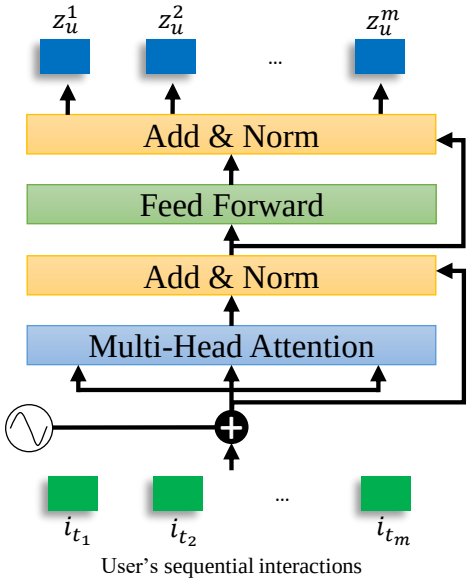


Figure 5: Transformer Encoder

C Pseudocode of training and inference

In our RecForest, all the trees share the encoder but each tree has a unique decoder. The training framework is summarized in Algorithm 3. Note that we only give one training instance for clear illustration and it can be done in batch-wise manner for practical implementation. In addition, the inference framework is summarized in Algorithm 4. Once the generated candidates from each tree are collected, we compute the generated probabilities of candidates on each tree for ranking purpose.

Algorithm 3: RecForest Training

Data: $Forest$=$\{tree_1, tree_2, \ldots, tree_T\}$, training instance (x_u, i), Encoder and Decoder$_t$ $(1 \leq t \leq T)$ w.r.t. each tree

$\boldsymbol{z}_u \leftarrow$ Encoder($\boldsymbol{x}_u$);
foreach $t \in \{1, 2, \ldots, T\}$ **do**
  target$\leftarrow$ get the path of item i on the $tree_t$;
  decoder-input$\leftarrow$ ['start',target[0:H-1]]; `// H is tree height`
  prediction$\leftarrow$ Decoder$_t$($\boldsymbol{z}_u$,decoder-input);
  $\mathcal{L}(u, i) \leftarrow$ loss(prediction, target[1:H]) according to Eq (1);
  back-propagation from $\mathcal{L}(u, i)$;
  update Encoder and Decoder$_t$ by gradient descent;

Algorithm 4: RecForest Inference

Data: $Forest$=$\{tree_1, tree_2, \ldots, tree_T\}$, Encoder and Decoder$_t$ $(1 \leq t \leq T)$ w.r.t. each tree, user feature x_u, beam size B, n (i.e. number of returning items)

Result: The top-n items for user u

$R \leftarrow \{\}$;
$\boldsymbol{z}_u \leftarrow$ Encoder($\boldsymbol{x}_u$);
for $t \in \{1, 2, \ldots, T\}$ **do**
  paths $\leftarrow$ Beam-Search(Decoder$_t$, $\boldsymbol{z}_u$, B) ;
  foreach $p \in$ *paths* **do**
    item $i \leftarrow$ get-item($p, tree_t$);
    $R \leftarrow R \cup \{i\}$;
foreach *item* $i \in R$ **do**
  $score[i] = 0$;
  for $t \in \{1, 2, \ldots, T\}$ **do**
    $path \leftarrow$ get-path($i, tree_t$);
    $p_t(i) \leftarrow$ compute the $path$ probability based on Decoder$_t$;
    $score[i] = score[i] + \log p_t(i)$;
select the top-n items from R based on $score$;

D Metrics

The ranking metric, Normalized Discounted Cumulative Gain($NDCG$), is used to measure the performance of recommendation engine. Discounted Cumulative Gain(DCG) can be calculated by

$$DCG@k = \sum_{i=1}^{k} \frac{2^{r_i} - 1}{\log_2(i+1)}$$

where r_i means the relevance score and k is the number of returned items. As we focus on implicit feedback, $r_i = 1$ if the test user really interacts with the returned i-th item otherwise $r_i = 0$. NDCG can be calculated by

$$NDCG@k = DCG@k / iDCG@k$$

where $iDCG$ means ideal DCG, i.e. ranking all the returned items according to relevance scores in a descending order and calculating the DCG on the ordered items. Additionally, we also concern the memory consumption of retrieval index and inference efficiency. Concretely, we record the model size of retrieval system and the time consumption in inference.

E Experiment Details in sequential recommendation scenario

E.1 Baselines in sequential recommendation scenario

- **TDM** [23] and its variant **JTM** [22], proposed by Alibaba Group, are the tree-based recommender frameworks which train the user-item preference model and the tree structure simultaneously. The

leaf nodes represent items and the internal nodes represent the clusters which the corresponding leaf nodes belong to. Beam search is conducted on the tree to retrieve the items from coarse-grain clusters to fine-grain clusters. The main difference between TDM and JTM is the updating of bijective mapping relation between items and leaf nodes.

- Deep Interest Network (i.e. **DIN**) [20] utilizes the deep model to characterize the user's preferences over items from user's sequential interactions. Concretely, DIN casts the training task as a binary-class classification that reflects whether the user interacted with certain item. As DIN can't be used for efficient recommendation, we report the brute-force-based performance.
- **YoutubeDNN** [1], proposed by Google, is a deep neural network model which learns the user representation and item representation from sequential user-item interaction history.Then the inner product between user representation and item representation is used to capture user's preference over the corresponding item, which converts the recommendation task to maximum inner product search(MIPS). In our experiments, we mainly show the performance of YoutubeDNN by burte-force search for MIPS.
- To solve MIPS efficiently, Quantizaiton-based method **SCANN** [3] and graph-based method **IPNSW** [10] meet the start-of-the-art performance in each technique route respectively. For fair comparison with our model, we model user's interactions by Transformer encoder and replace the decoder by item embedding. The adapted model is trained in the same way as DIN to obtain the user representation and item representation so that the learned representation can be used to build the SCANN index and IPNSW index.

The open-source code of TDM[2], JTM[3] and DIN[4] are based on Alibaba Group's X-DeepLeraning framework, we re-implement them by Pytorch. And the open-source code of YoutubeDNN[5] is also available.

E.2 Detailed settings in sequential recommendation scenario

In each dataset, we randomly choose 10% users as validation users, 10% users as test users and all the left users as training users. Following the settings in TDM [23] and JTM [22], we use a slide window to cut user-item interaction histories into slices of length at most 70. Each window contains at least 15 interactions and zero padding is applied if there is less than 70 interactions in the window. At the training stage, the first 69 user-item interactions are regarded as context and the 70-th item is regarded as ground truth in each window. For validation users and test users, we regard the second half interactions as ground truth and the first half interactions as context. The hidden size is set to 96 for all methods.

The Deep Interest Network (DIN) [20], which contains the embedding layer followed by a MLP with size [128,64,2] and outputs the like/dislike probability, is used as the the preference model for TDM and JTM. YoutubeDNN [1] contains embedding layer followed by a MLP with size [128,64,96] and the number of negative sampling is set to 1,000. For SCANN, we follow the default settings of the open sources[6] but implement them with Python and Pytorch for fair efficiency comparison. Concretely, the number of leaf is 2,000; the number of sub-space is 24 and the dimensionality of each sub-space is 4; the number of codewords in each sub-space is set to 16; the value of threshold is set to 0.2. To construct the graph index of IPNSW, the maximum degree of each node is set to 16, and the beam size is set to 100.

For our RecForest, YoutubeDNN, TDM, JTM and DIN, Adam [7] is used as the optimizer and learning rate is set to 1.0e-3 with exponential decay. In inference, the beamsize of RecForest, TDM, JTM and IPNSW is set to 100 unless specification. For RecForest, the branch number is set to 18 and the tree number is set to be 2 on Movie and MIND, and 4 on the left datasets unless specification. The items' representations which are utilized to construct the forest for RecForest are obtained by pre-train the DIN model on each dataset.

[2] https://github.com/alibaba/x-deeplearning/tree/master/xdl-algorithm-solution/TDM

[3] https://proceedings.neurips.cc/paper/2019/hash/1c6a0198177bfcc9bd93f6aab94aad3c-Abstract.html

[4] https://github.com/alibaba/x-deeplearning/tree/master/xdl-algorithm-solution/DIN

[5] https://github.com/shenweichen/DeepMatch

[6] https://github.com/google-research/google-research/tree/master/scann

F Experiments in non-sequential scenarios

In non-sequential scenarios, we only apply user id to denote a user and the feature encoder to model user representation. Specifically, the learnable embedding vector of the user id denotes user representation.

F.1 Baselines

The classical deep model, Neural Collaborative Filtering (**NCF**) [4], characterizes user's preference over item given the user id and item id. Due to the compatibility with any advanced user-item (or user-node) preference model, we utilize NCF for preference modeling in **TDM** [23] and **JTM** [22] frameworks. For **YoutubeDNN**, we replace the user representation learning part used in sequential recommendation scenarios with the embedding vector of user id. By this way, YoutubeDNN can be compatible with non-sequential recommendation task. **SCANN** and **IPNSW** are built on the learned representation of YoutubeDNN. Note that brute-force-based YoutubeDNN and NCF are time-prohibitive in online services, but they can be considered as two strong baselines.

F.2 Experimental settings

In each dataset, we randomly choose 10% users as validation users, 10% users as test users and all the left users as training users. For validation users and test users, we randomly select a quarter of the interacted items as ground truth, and put the rest into the training set. The fusion NCF [4], which contains the GMF and MLP where the neuron numbers are set to [512,256,128, 96] in each layer, is included. Other necessary settings are as described in Section E.2.

F.3 Comparison with Baselines

Settings We compare RecForest with the baselines on all datasets. RecForest uses 2 trees on the Movie and MIND, and 4 trees on other datasets. Other settings can be referred to in Section F.2.

Results All the results are shown in Table 6, where the index memory cost of RecForest indicates the decoder's memory consumption. From the table, we have the following findings.

- **Recforest nearly outperforms all efficient recommenders with indexes on all dataset w.r.t. NDCG@20 and NDCG@40** except that SCANN outperforms RecForest slightly on Movie w.r.t NDCG@40. Its superiority over two-stage baselines is not so significant indicates that inner product may be capable enough for representation learning to some extent in this simple recommendation scenario. The higher accuracy than TDM and JTM also verify the benefit from the use of routing trajectory, multiple trees and the powerful transformer decoder.
- **RecForest performs better than brute-force-based NCF and YoutubeDNN on nearly all datasets** except the results on Tmall where YoutubeDNN outperforms RecForest slightly. This phenomenon confirms the the superpower of transformer decoder.
- Being similar to the results of sequential recommendation scenarios, **RecForest has the advantage in index memory cost**, which can be highlighted by comparisons with baselines. These observations can be explained by referring to the same experiments in sequential scenarios. Note that NCF and YoutubeDNN are based on exhaustive search so they don't consume memory for indexes.

F.4 Extensive Study between Efficiency and Accuracy

Settings Similar to settings in sequential recommendation scenarios, we also draw the efficiency-accuracy curve. The study is mainly investigated on the Amazon, MIND, Gowalla, and Yelp dataset, since the other two datasets show a similar trend. As described in the sequential scenario, we adjust the beams size to vary the retrieval time cost for TDM, JTM, IPNSW and RecForest. For SCANN, we adjust the number of probe cells to control the time cost.

Results The curves of different algorithms between NDCG@20 and query time are shown in Figure 6. From the figure, we have the following findings.

Table 6: Comparison with Baselines w.r.t NDCG@20 and NDCG@40, index memory space (MB), and inference time (second). The bold fonts indicate the best performance.

	NDCG@20	NDCG@40	Memory	Time	NDCG@20	NDCG@40	Memory	Time
Method	**Movie**				**Amazon**			
NCF	0.1679	0.2252	-	52.61	0.0433	0.0703	-	447.30
YoutubeDNN	0.2002	0.2532	-	39.98	0.1004	0.1284	-	129.57
JTM	0.2142	0.2535	10.80	1.62	0.0544	0.0799	75.99	2.57
TDM	0.2164	0.2536	10.80	1.62	0.0251	0.0322	75.99	2.43
SCANN	0.2227	**0.2646**	3.64	1.93	0.0936	0.1168	14.66	2.56
IPNSW	0.2014	0.2014	10.08	3.64	0.1022	0.1124	66.46	2.80
RecForest	**0.2229**	0.2640	**3.21**	**1.50**	**0.1064**	**0.1338**	**7.32**	**2.20**
Method	**Gowalla**				**Tmall**			
NCF	0.1106	0.1545	-	304.79	0.0340	0.0490	-	3039.39
YoutubeDNN	0.3494	0.3613	-	52.97	0.0820	0.1005	-	1158.76
JTM	0.1404	0.1512	77.56	2.00	0.0722	0.0880	151.19	4.49
TDM	0.1462	0.1562	77.56	2.09	0.0596	0.0736	151.19	5.36
SCANN	0.2078	0.2179	15.48	1.96	0.0650	0.0754	28.10	5.45
IPNSW	0.3472	0.3589	70.39	1.98	0.0671	0.0707	132.72	5.20
RecForest	**0.3500**	**0.3651**	**7.39**	**1.91**	**0.0743**	**0.0905**	**9.29**	**3.85**
Method	**MIND**				**Yelp**			
NCF	0.4070	0.4392	-	21.20	0.1078	0.1471	-	93.59
YoutubeDNN	0.4194	0.4428	-	19.31	0.1633	0.1944	-	55.82
JTM	0.3476	0.3664	6.62	3.34	0.1262	0.1636	39.47	3.36
TDM	0.3655	0.3799	6.62	3.25	0.0472	0.0583	39.47	3.42
SCANN	0.3799	0.3822	3.20	3.45	0.1250	0.1523	8.44	4.10
IPNSW	0.4178	0.4399	6.95	3.85	0.1641	0.1942	34.74	3.59
RecForest	**0.4308**	**0.4587**	**3.18**	**3.19**	**0.1720**	**0.2016**	**6.81**	**3.24**

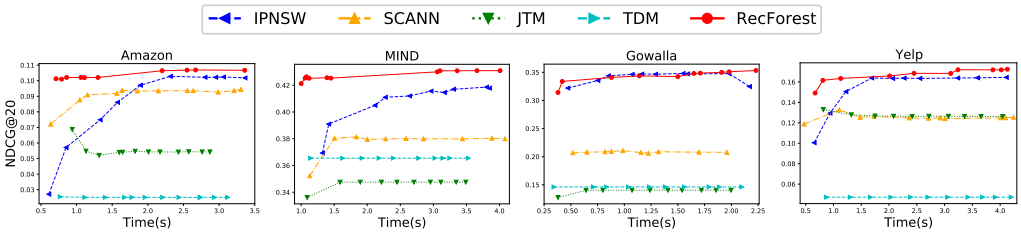


Figure 6: Tradeoff between Efficiency and Accuracy

- **RecForest strikes the best balance between query time and retrieval accuracy on all the four datasets**, since the curve stands on top of the others. This confirms the superiority of RecForest to these competing baselines. The advantage is the most significant on the MIND, evidenced by the biggest gap between RecForest and IPNSW.
- **With the increase of beam size, the accuracy of RecForest can improve more significantly than baselines.** This benefits from the powerful transformer decoder and the novel training paradigm as well as the forest-based index, such that the representation model and tree indexes are jointly learned better.

F.5 Ablation Study

F.5.1 Effect of Forest Construction

Settings We investigate three aforementioned ways of constructing a forest with at most 10 trees, where the branch number of each tree is set to 4. The details about forest construction can be referred to in Section 2.3.2.

Results The results on the Amazon, MIND, Gowalla and Yelp datasets are reported in Figure 7. The results on Amazon, MIND and Yelp show the same trends as ones in sequential scenarios. Concretely, **Kmeans with randomness (i.e. Random+Kmeans) always performs best and improves with the increasing number of trees while the Random nearly performs worst and does not improve as much as Random+Kmeans when the tree number increases.** These results verify the findings again, i.e. the semantic information and the diversity among trees are important to improve the recommendation accuracy.

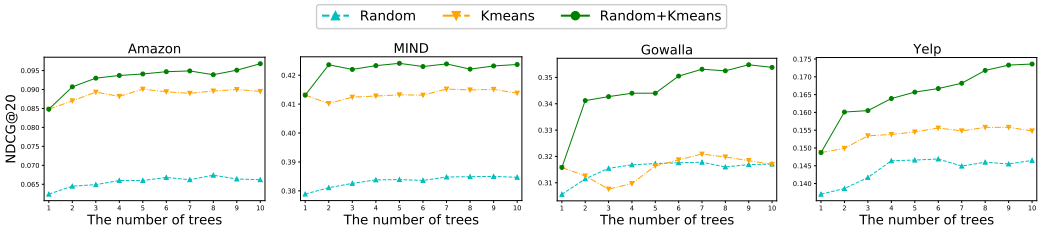


Figure 7: Ablation study: Different construction ways for forests.

F.5.2 Effect of Branch Number K

Settings We conduct evaluations on a single Kmeans-constructed tree on the Amazon, MIND, Gowalla and Yelp dataset, where the branch number varies in {2,4,8,10,16,18}.

Results The results are shown in Table 7. We can observe that **the retrieval accuracy of RecForest on Amazon improves with the growing number of branches constantly but does not show a clear trend on other three datasets.** These findings indicate that the sensitivity of branch number depends on the dataset in non-sequential recommendation scenarios. Generally, the increase of branch numbers and the decrease of path length don't affect the performance as much as that in the sequential scenario.

Table 7: Effect of branch number

#Branch	NDCG@20			
	Amazon	MIND	Gowalla	Yelp
2	0.0724	0.4028	0.2916	0.1538
4	0.0869	0.4064	0.2837	0.1502
8	0.0886	0.3964	0.2947	0.1531
10	0.0913	**0.4281**	0.2949	0.1520
16	0.0961	0.4005	**0.3052**	**0.1562**
18	**0.0971**	0.4107	0.2978	0.1555

F.5.3 Effect of Positional Embedding

Settings We also investigate the effect of four positional embedding methods like what we do in the sequential recommendation scenario. Concretely, (1)**None**;(2)**Abs**;(3)**RelK**;(4)**RelKQ** and more details are referred to in Section 3.6.3.

Results The results based on a single Kmeans-constructed tree on the Amazon, MIND, Gowalla, and Yelp datasets are summarized in Table 8. It is similarly observed to Section 3.6.3 that **positional embedding takes a remarkable effect on improving the retrieval capability while different positional embedding methods do not have a significant difference in retrieval accuracy**. The effect of Abs and RelK is more significant on Gowalla.

Table 8: Effect of positional embedding

Pos. Emb.	NDCG@20			
	Amazon	MIND	Gowalla	Yelp
None	0.0629	0.3656	0.2028	0.1140
Abs	**0.0869**	**0.4064**	0.2837	0.1502
RelK	0.0719	0.3975	**0.2878**	**0.1509**
RelKQ	0.0653	0.4049	0.2436	0.1443

G Additional Ablation Studies

To further verify the effect of Transformer Decoder and the joint training between the representation model and the ANN index, we conduct additional ablation studies.

G.1 Effect of Transformer Decoder

Settings We conduct evaluations on a single Kmeans-constructed tree with 4 branches, and replace the Transformer Decoder with the RNN and CNN model. The hidden size and embedding size of RNN and CNN are set to be 96. Additionally, the kernel size of CNN is set to be 3.

Results The results are summarized in Table 9. We can see that the Transformer Decoder always performs best and the performance gaps between them are remarkable. This verifies the superiority of the Transformer Decoder for sequence generation tasks in our model compared to CNN and RNN.

Table 9: Effect of Transformer Decoder

Decoder	NDCG@20			
	Amazon	MIND	Gowalla	Yelp
Transformer	**0.1912**	**0.6935**	**0.2795**	**0.2269**
RNN	0.1208	0.6912	0.2286	0.1984
CNN	0.1218	0.6933	0.1758	0.1691

G.2 Effect of Joint Training

Settings To reveal the impact of joint representation learning, we include another experiment, where the joint representation learning is disabled. To be specific, we initialize the index parameter, i.e., the embedding of each branch, as the average of all the item embeddings within the corresponding branches. The parameters of the index are fixed during the training process, so that only the parameters of Encoder and Decoder are learnable. Such a variant is short for "Fixed Index" in our experiment.

Results The results are summarized in Table 10. The performance of fixed index is worse than the original one, which indicates the removal of joint representation learning indeed degrades the capability of our model. In other words, joint training is indeed important and necessary.

Table 10: Effect of Joint Training

Index	NDCG@20			
	Amazon	MIND	Gowalla	Yelp
Joint Training	**0.1912**	**0.6935**	**0.2795**	**0.2269**
Fixed Index	0.1429	0.6891	0.1604	0.1588