RecStudio: Towards a Highly-Modularized Recommender System

Defu Lian*
liandefu@ustc.edu.cn
University of Science and Technology of China

Xu Huang
xuhuangcs@mail.ustc.edu.cn
University of Science and Technology of China

Xiaolong Chen
chenxiaolong@mail.ustc.edu.cn
University of Science and Technology of China

Jin Chen
chenjin@std.uestc.edu.cn
University of Electronic Science and Technology of China

Xingmei Wang
xingmeiwang@mail.ustc.edu.cn
University of Science and Technology of China

Yankai Wang
echobelbo@mail.ustc.edu.cn
University of Science and Technology of China

Haoran Jin
HaoranJin@mail.ustc.edu.cn
University of Science and Technology of China

Rui Fan
jennahfr@mail.ustc.edu.cn
University of Science and Technology of China

Zheng Liu
zhengliu1026@gmail.com
Huawei

Le Wu
lewu.ustc@gmail.com
Hefei University of Technology

Enhong Chen
cheneh@ustc.edu.cn
University of Science and Technology of China

ABSTRACT

A dozen recommendation libraries have recently been developed to accommodate popular recommendation algorithms for reproducibility. However, they are almost simply a collection of algorithms, overlooking the modularization of recommendation algorithms and their usage in practical scenarios. Algorithmic modularization has the following advantages: 1) helps to understand the effectiveness of each algorithm; 2) easily assembles new algorithms with well-performed modules by either drag-and-drop programming or automatic machine learning; 3) enables reinforcement between algorithms since one algorithm may act as a module of another algorithm. To this end, we develop a highly-modularized recommender system – *RecStudio*, in which any recommendation algorithm is categorized into either a ranker or a retriever. In the *RecStudio* library, we implement 90 recommendation algorithms with the pure Pytorch, covering both common algorithms in other libraries and complex algorithms involving multiple recommendation models. *RecStudio* is featured from several perspectives, such as index-supported efficient recommendation and evaluation, GPU-accelerated negative sampling, hyperparameter learning on the validation, and cooperation between the retriever and ranker. *RecStudio* is also equipped with a web service, where the recommendation pipeline can be quickly established and visually evaluated on selected datasets, and the evaluation results are automatically archived and visualized in a leaderboard. The project and documents are released at http://recstudio.org.cn.

CCS CONCEPTS

• **Information systems** → **Recommender systems**.

KEYWORDS

Recommender System, Modularization, Web Services, Multi-Stage

ACM Reference Format:
Defu Lian, Xu Huang, Xiaolong Chen, Jin Chen, Xingmei Wang, Yankai Wang, Haoran Jin, Rui Fan, Zheng Liu, Le Wu, and Enhong Chen. 2023. RecStudio: Towards a Highly-Modularized Recommender System. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23), July 23–27, 2023, Taipei, Taiwan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3539618.3591894

1 INTRODUCTION

Recommender systems provide an essential way to alleviate information overload issues. These techniques not only improve user experiences in diverse areas like e-commerce, online education, and personal assistance, but also create great value for many high-tech companies, such as Amazon, Google, Microsoft, and Taobao. As a consequence, recommender systems have been a long-standing research topic, producing many algorithms from both academia and industry. Recent efforts have been devoted to develop unified and reproducible frameworks [67, 85, 89] with standardizing inputs, model interfaces, and evaluation for accommodating popular recommendation algorithms. These frameworks have been implemented with many different programming languages like C++, Java, Python, Matlab, and C#, and even different python deep learning libraries, like TensorFlow and PyTorch. They have significantly accelerated the development of open-source recommender systems.

*Corresponding author

SIGIR '23, July 23–27, 2023, Taipei, Taiwan

ACM ISBN 978-1-4503-9408-6/23/07…$15.00
https://doi.org/10.1145/3539618.3591894

We have extensively investigated these frameworks, and observed the following limitations:

• **These frameworks are usually designed for research purposes, barely considering their practical usage in real scenarios**. It is of low efficiency to conduct the evaluation on the held-out datasets in most frameworks, since they have to make inferences for each candidate item, even though the inference is implemented by C++/C. A practical scalable recommender system usually calls for a multi-stage workflow of cascade ranking, but there is currently no framework to bring them together. This also leads to the gap between academia and industry, where the algorithms from academia are difficult to test/deploy in the industry while the algorithms from the industry are challenging to compare as baselines.

• **These frameworks are almost simply a collection of algorithms, overlooking the modularization of recommendation algorithms.** The implementation of algorithms only follows standard model interfaces, such that the evaluation protocol can be standardized. However, the recommendation algorithms have almost identical architectures and share many common components like data augmentation [77], negative sampling [46, 82], scoring functions, encoders of sequence [39]/KG [80], feature interaction operators [24, 87], debiasing [66] and loss functions. This squanders the chance of creating better recommendation algorithms by combining the best modules in each component.

To address these issues, we have initiated a project called *RecStudio* for developing highly-modularized recommender systems. In addition to the commonly-concerned reproducibility of existing models and the standardization of evaluation protocol, *RecStudio* adds new features for further facilitating the implementation of existing algorithms or the development of new algorithms by disassembling the recommendation algorithms into reusable modules. These key features and capabilities of *RecStudio* are summarized in the following six perspectives.

• **Modularizing recommendation models.** Though hundreds of recommendation algorithms were proposed, these algorithms only differ in small parts, like loss functions and feature encoders. Therefore, it is essential to disassemble the recommendation algorithms into small reusable blocks called modules, such that it is less error-prone yet more convenient to implement existing algorithms and assemble new excellent algorithms with these modules by either drag-and-drop programming or automatic machine learning.

• **Assembling both retrievers and rankers.** A scalable recommender system usually calls for a multi-stage workflow, as demonstrated in Figure 2. Within the workflow, a retriever first selects a small set of candidates from the entire items with high efficiency and the cascade rankers are then used to refine the best items from the retrieval results with highly expressive yet time-consuming networks. These different requirements lead to their use of different architectures. *RecStudio* assembles and unifies both the retriever and ranker, and implements the cascade ranker by embedding a retrievable ranker with the top-k functionality. This makes it possible to scale online recommendations for massive items and to optimize the entire multi-stage workflow simultaneously.

• **Supporting efficient recommendation and evaluation with ANNs indexes and multi-stage filters.** The retriever is equipped with an ANNs index [38], which is either incrementally updated or rebuilt from scratch every several epochs, such that the

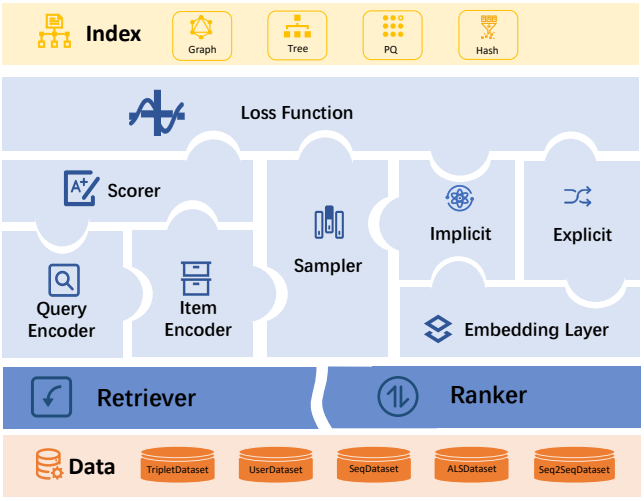

Figure 1: The framework of *RecStudio*.

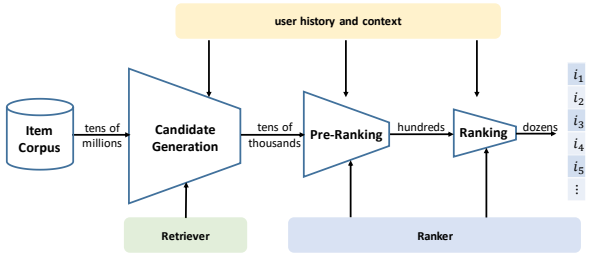

Figure 2: Multi-stage workflow of scalable recsys.

top-k results can be retrieved from the retriever in sublinear time. Based on the top-k results, *RecStudio* implements a novel efficient method to compute the ranking metrics on the validation/testing set with pure tensor operators. When embedding a retriever inside, the ranker also supports the top-k functionality, which is called the retrievable ranker. The retrievable ranker returns the top-k results by retrieving a small set of candidates from the retriever and refining the retrieval results based on the ranker's inference.

• **Accelerating negative sampling with GPU.** Both the retriever and the retrievable ranker are usually trained on implicit feedback, which is positive only and unlabeled. Therefore, negative samples have to be drawn from unlabeled data. The negative sampling methods have evolved from static sampling (e.g. uniform sampling [62] or frequency-based sampling [61]) to model-based dynamic sampling (e.g. dynamic negative sampling [82], cluster-based sampling [46], or LSH sampling [68]). Though static sampling is more efficient, it usually leads to slow convergence due to the large divergence between sampling distributions and the real distribution. Existing libraries almost draw negative samples when preparing the datasets, so it is challenging for them to adopt model-based dynamic sampling methods. *RecStudio* treats negative samplers as a module of recommendation algorithms and implements both static sampling and dynamic sampling with pure tensor operators so that negative sampling can be accelerated with GPU.

• **Learning hyperparameters on the validation set.** The hyperparameters are usually tuned on the validation set, by exploring

Table 1: Score Functions

Score function	Formula
Inner Product [32, 62]	$r = \mathbf{q}^T \mathbf{i}$
Cosine Similarity [54]	$r = \frac{\mathbf{q}^T \mathbf{i}}{\|\mathbf{q}\|\|\mathbf{i}\|}$
Squared Euclidean distance [27, 31]	$r = \|\mathbf{q} - \mathbf{i}\|_2^2$
MLP [30]	$r = (f_{\theta_n} \circ \cdots \circ f_{\theta_0})([\mathbf{q}; \mathbf{i}])$
ℓ_p-Norm[2, 81]	$r = \|\mathbf{q} - \mathbf{i}\|_p$
GMF [30]	$r = \sigma(\mathbf{q} \odot \mathbf{i})$
GMFMLP [30]	$r = \sigma(\mathbf{h}^T a(\mathbf{q} \odot \mathbf{i} + \mathbf{W}[\mathbf{q}; \mathbf{i}] + \mathbf{b}))$

the space for good values. The search algorithms can be grid search, Bayesian optimization, evolutionary algorithms, or reinforcement learning, which are integrated into most hyperparameter tuning libraries like Ray Tune [51], Hyperopt [5], Scikit-Optimize [70], Microsoft NNI [56], and so on. In addition to hyperparameter tuning, *RecStudio* also integrates the learning of some hyperparameters on the validation set through (stochastic) gradient descent, based on the gradient of validation loss with respect to the hyperparameters. The learning of hyperparameters is much more efficient than tuning, but it does not apply to all hyperparameters.

• **Jointly optimizing the retriever and cascade rankers.** In addition to assembling both retrievers and rankers, *RecStudio* moves a step further, implementing all hitherto known joint optimization algorithms between the retriever and cascade rankers. It realizes cascading a retriever or a retrievable ranker before the ranker and provides them with an interface of negative sampling and top-k retrieval. In particular, the retriever may generate hard negatives for the rankers' training, while the rankers could transfer their knowledge of ranking hard negatives to the retriever. This enables a bidirectional information flow between the retriever and rankers.

2 RECSTUDIO DESIGN

The framework of *RecStudio* is illustrated in Figure 1, where recommendation models are disassembled into several modules. We will introduce the modules as follows.

2.1 Input Design

The flow from the raw data format to the model input involves the following steps: (1) load data configuration (2) pre-process raw data (3) split data for training and testing (4) build mini-batches. Four dataset structure classes, i.e., *TripletDataset, UserDataset, SeqDataset, Seq2SeqDataset and ALSDataset*, are carefully designed in this library to support all mainstream recommendation tasks, which vary in the input format of the mini-batch data. Arbitrary raw datasets in the `.csv` and `.tsv` format support these data structures to implement arbitrary recommenders and we collect popular 10 datasets in the latest version. Our library has the following three careful designs for input to make it much easier to get started, more convenient and more efficient.

User-friendly Configuration. The configuration of datasets is particularly convenient in this brand-new library, which only requires setting a few essential fields. These fields can be configured flexibly. Currently, we support command line input, YAML format configuration, python dictionary input, or importing configuration files via the visual front-end page. Once the major fields are set, the library automatically performs the data pre-processing, including data filtering, missing value filling, id mapping, data split, etc., in preparation for subsequent model learning.

Time-saving File Reading. Considering the general scenario where a single dataset will be experimented with multiple times, including different data splits and different recommendation algorithms, the library is designed with a caching switch to store the processed files in a binary file format for subsequent reading directly from the disk into memory. In particular, given the considerably large-scale dataset, the saved caching avoids the time-consuming pre-processing procedure except for the first loading, such as data filtering and id mapping, which will consume a large amount of IO, and thus reduces the waiting time for data preparation.

Highly Parallelized Data Loader. Building multiple mini-batches from the entire dataset for training requires a large amount of communication, especially in the scenario of sequential recommendations with large-scale datasets. In this way, we overwrite the sampler class to achieve more efficient loaders. Specifically, within each mini-batch, the sampler returns a batch of indexes rather than a single index at once, which can be easily accomplished thanks to the intelligent slicing of tensors. Compared to the native multi-process loader, we empower the library to perform a highly automated parallelization with no user perception and with no need to manually set the process numbers.

2.2 Model Design

2.2.1 General Design. After extensive surveys of current recommendation models, we summarize and decouple the whole learning schema into the following components: mapping function to encode the raw input, score function to predict the user preference, and loss function to guide model learning. The *mapping function* is determined by the current mainstream recommendation tasks, presenting the encoders for retrievers and consisting of feature embedding and interaction layers for rankers and we will introduce them in the next section. *scorer function* usually calculates the similarity between the given query and item and we currently implement seven scoring functions as shown in Table 1. As for the *loss function*, it calculates the deviation from the ground-truth labels and then guides the model learning. We categorize existing loss functions in recommendation models into three base classes, i.e., *ListwiseLoss, PariwiseLoss and PointwiseLoss*, as shown in Table 2. By integrating different loss functions, the multi-task learning is easily implemented. In this way, we disassemble the implementation of recommendation algorithms into reusable blocks to realize the unified interface for existing models to avoid duplicate coding and facilitate the implementation of user-defined models. Specifically, we are able to quickly implement most of the currently available models by concatenating the modules and making appropriate substitutions for each module, making the whole process as simple as building blocks. In addition to the general processes described above, *RecStudio* supports the following extensions, which cover all directions of current research in recommender systems.

• **Contrastive Learning** first draws positive and contrastive samples from the entire corpus or from the mini-batch, depending on which the contrastive loss is calculated. We have implemented the popular augmentation strategies as shown in Table 3, e.g., ItemCrop, Mask, Substitution, Insertion, Reorder, Feature Clustering, for sequential recommenders and EdgeDrop, NodeDrop, Feature

Table 2: Loss Functions

loss	Formula	Type	Complexity	Related Metric
SoftmaxLoss[16]	$L = -\log \frac{\exp f_\theta(c,k)}{\sum_{i=1}^{N} \exp f_\theta(c,i)}$	Listwise	$O(N)$	NDCG
BPRLoss[62]	$L = -\frac{1}{\lvert S\rvert} \sum_{i \in S} \log \sigma(f_\theta(c,k) - f_\theta(c,i))$	Pairwise	$O(\lvert S\rvert)$	AUC
Top1Loss	$L = -\frac{1}{\lvert S\rvert} \sum_{i \in S} (\sigma(f_\theta(c,i) - f_\theta(c,k)) + \sigma(f_\theta(c,i)^2))$	Pairwise	$O(\lvert S\rvert)$	-
BinaryCrossEntropyLoss[28, 39]	$L = -(\log \sigma(f_\theta(c,k)) + \sum_{i \in S} \log(1 - \sigma(f_\theta(c,i))))$	Pairwise	$O(\lvert S\rvert)$	logloss
HingeLoss[31]	$L = -f_\theta(c,k) + (f_\theta(c,j) - \text{margin})$	Pairwise	$O(1)$	AUC
SampledSoftmaxLoss[7, 13]	$L = -\log \frac{\exp(f_\theta(c,k) - \log Q(k\lvert c))}{\sum_{i \in S \cup \{k\}} \exp(f_\theta(c,i) - \log Q(i\lvert c))}$	Listwise	$O(\lvert S\rvert)$	NDCG
InfoNCELoss	$L = -\log \frac{\exp(f_\theta(c,k))}{\sum_{i \in S \cup \{k\}} \exp(f_\theta(c,i))}$	Listwise	$O(\lvert S\rvert)$	DCG
BCEWithLogitLoss[24, 47, 59]	$L = -\big(y_k \log \sigma(f_\theta(c,k)) + (1 - y_k) \log(1 - \sigma(f_\theta(c,k)))\big)$	Pointwise	$O(1)$	logloss
MSELoss[12]	$L = -(y_k - f_\theta(c,k))^2$	Pointwise	$O(1)$	MSE

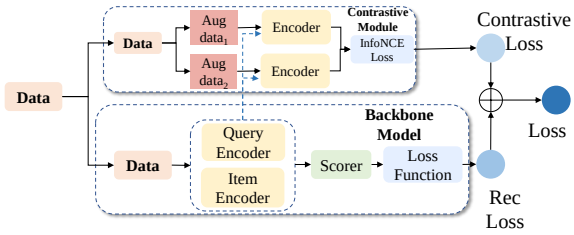

Figure 3: Contrastive Learning Framework in *RecStudio*.

Table 3: Contrastive Learning Models

Model	Data Augmentation	Model Type
CL4SRec [77]	ItemCrop, Mask, Reorder	Sequential
ICLRec [14]	Feature Clustering	Sequential
CoSeRec [53]	Substitution, Insertion	Sequential
SGL [76]	EdgeDrop, NodeDrop	Graph-based
NCL [52]	Feature Clustering, Neighbor Aggregation	Graph-based
SimGCL [79]	Random Noise to hidden representations	Graph-based

Clustering, Neighbor Aggregation, Random Noise for graph-based recommenders, and support the InfoNCE objective functions. Specifically, we design the following flow, as shown in Figure 3, to integrate the contrastive learning module into the whole modularized *RecStudio*. By selecting data augmentation methods in the contrastive module, a contrastive loss based on InfoNCE would be added to the recommendation loss for the training procedure.

• **Debiasing Learning** is affiliated with backbone models and these debiasing methods alleviate the bias by modelling different tasks respectively or assigning inverse propensity scores to samples. The whole framework is illustrated in Figure 4. By keeping the backbone unchanged but adding some additional modules or functions, it is of great convenience to figure out how debiasing methods perform across different backbones, instead of re-implementing. And until now, *RecStudio* is the only and first library for freely debiasing backbones and we now support 9 common debiasing methods.

2.2.2 Retriever. Retriever models attempt to extract a subset of items with high potential preferences from all candidates, under the constraints of quick response. A common paradigm of retrievers

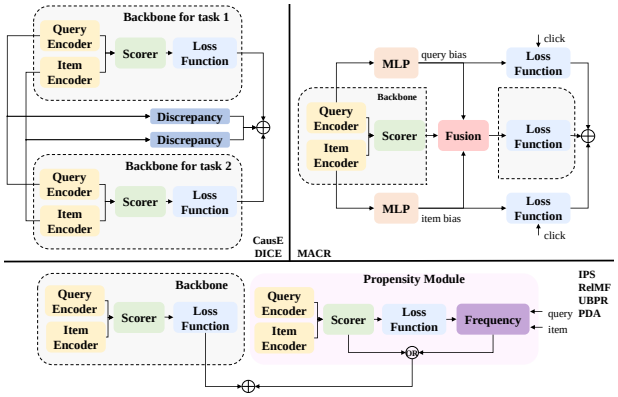

Figure 4: Debiasing Learning Framework in *RecStudio*.

Table 4: Debiasing Models

Model	Issue	Solution
IPS [66]	Selection bias	Propensities added to loss
RelMF [64]	Selection bias Positive-Unlabeled	Propensities added to loss
UBPR [63]	Selection bias Positive-Unlabeled	Propensities added to loss
CausE [8]	Selection bias	Missing-At-Random data
DICE [86]	Conformity bias	Disentangle task-specific embeddings
PDA [83]	Popularity bias	Intervene
MACR [74]	Popularity bias	Counterfactual reason
ExpoMF [49]	Selection bias	EM algorithm
IPW [48]	Selection bias	Propensities added to loss

includes several steps: (1) give training samples, usually positive only feedback, (2) sample negatives from unlabeled items according to static distributions or model-based distributions, (3) encode the user query according to the user's contextual information or user history through complex neural networks and encode the item through a special neural network, (4) calculate the loss function and optimize the model parameters, (5) search for the top-k relevant items inference. The general algorithm libraries implement steps

Table 5: Samplers in *RecStudio*

Type	Sampler	Examples
Static	Uniform	BPR[62], SASRec[39]
	Popularity-based	AOBPR[61]
	Inbatch	CL4SRec[77], G-Tower[78]
Model-based	DNS	DNS[82]
	Reject Sampling	CML[31], WARP[75]
	LSH Sampling	MONGOOSE[11], LSH-PFE[68]
	Cluster Sampling	PRIS[46]
	MIDX Sampling	FastVAE[13]

1-2 during the data pre-process phase (data loader), usually on the CPU, and then convert the data to the GPU for steps 3-4, after which traverse the whole item candidates for step 5. Nevertheless, we creatively implement GPU-based samplers and access to ANN search to meet the requirements of both efficiency and accuracy.

Negative sampling has evolved from static sampling to model-based sampling to achieve more efficient convergence. Static sampling methods, including uniform and frequency-based sampling, are independent of the model change, and as a result, sampling has been integrated into data pre-processing in existing libraries. Despite their high efficiency, static sampling approaches are prone to slow convergence due to the huge deviation between the sampling distribution and the real distributions. The dynamic model-based samplers, where the sampling distribution is reliant on the predicted preference, assist in fast convergence but are not well supported in the pre-processing step. Data transfer from memory to GPU consumes a significant amount of IO, and thus prevents the efficient model-based sampling process. *RecStudio* introduces a single *sampling* module, where the negative sampling is treated as a PyTorch module, to implement both the static and dynamic model-based samplers, with the aim of accelerating the sampler with GPU. Specifically, we utilize the result of query encoding as the input of the sampler and calculate the corresponding sampling probability for each query, which can be obtained directly from the PyTorch tensor operation. The sampler then returns the indexes of items with the sampling probability based on the calculated probability. The entire process is deployed on the GPU, remarkably accelerating the computation of probabilities, rather than in a CPU-based data pre-processing step as in existing solutions. Furthermore, we unify the interface, including initialization, update with epoch/batch increasing and sampling, which enables most sampling strategies to be implemented. Up to now, we have implemented eight samplers in *RecStudio*, as illustrated in Table 5, involving both static and model-based dynamic samplers.

Another key feature for retrievers is that we have assembled fast top-k research in *RecStudio* to provide the retrieved ranking list, which is commonly implemented by enumerating all candidate items in existing libraries. With the increasing number of candidate items or incrementally updated items, trivial solutions require the calculation from the scratch, such as rebuilding the index, which takes much more cost. To facilitate the efficient search, we equip *RecStudio* with the interface for fast ANN search, which can support third-party fast search with GPU, e.g., FAISS[38], SCANN [25] and BLISS[26] or a custom index structure, e.g., tree-based [19, 20, 88] or graph-based indexes. These GPU-based ANN indexes enable fast top-k research to achieve the high efficient inference.

2.2.3 Ranker. The ranker aims to perform a more accurate ranking on the top-k items returned by retrievers, which often takes all data as input into a simple and big network to embed the interaction relationships between the fields. This leads to the ranker not having a separate item encoder like the retriever. The ranker is usually decomposed into several parts, including the dense embedding module, the explicit interaction module and the DNN module. We have implemented FM layer [24, 59], CrossNet layer [73], DIN layer [87], CIN layer [47] and MLP layer [15] for the high-order interaction module to model high-order interaction between sparse features. These modules can be flexibly combined, including horizontal concatenation, such as FM and MLP combined into DeepFM, or vertical stacking, such as DNN networks. Depending on these layers, we have implemented 29 rankers in Recstudio.

In addition, we support the cascading rankers, where the ranker module consists of several independent rankers cascaded together, as shown in Figure 2. Each ranker filters fewer preferred items according to the top-k ranked results for the next ranker and has its own learning parameters. Cascading rankers can be updated by two methods, including the multi-stage workflow and joint learning, and further exposition is in the following section. Such cascading rankers provide increasingly accurate ranking lists with more expressive and elaborate networks and thus improve the overall recommendation quality.

2.3 Model Training

Most existing recommendation algorithm libraries generally implement a **single model** training, commonly including the process of forward computation and backward update. Considering the real and complex deployment under industrial scenarios, which typically consist of at least two categories of models, i.e., retriever and ranker, to improve the recommendation quality, *RecStudio* is designed to enable the **multi-stage workflow** to get closer to the real industrial scenarios and provide the potential for the connection of the academy and industry. Specifically, the retriever extracts a set of items from the entire candidates with extremely high efficiency, followed by rankers reranking the items with more accurate but time-consuming networks. All models are optimized separately, but with different input from the last model. That is to say, the current model is trained according to the output of the previous model and only affects the input of the next following model. *RecStudio* enables such training paradigms simultaneously and allows to scale to online recommender systems for numerous items.

Furthermore, **joint optimization for retrievers and rankers** is equipped with *RecStudio*, where retrievers and rankers are influenced by each other. To be specific, we design the interface of negative sampling, which provides hard negatives for both retrievers and rankers, and the interface of top-k retrieval for inference. Rather than a rank-oriented loss, the retriever is additionally optimized by a distilled loss from the more expressive ranker to learn more precise results, realizing the bidirectional workflow for training retrievers and rankers. All hitherto known joint optimization algorithms, including Rankflow [57], and CoRR [33], are implemented in *RecStudio*. All methods can be easily integrated into *RecStudio* depending on the unified interface by varying the objective loss, where both retrievers and rankers are updated within the

same epoch. Such joint optimization workflow may shed new light on industrial deployments for more accurate ranking performance.

The multi-stage workflow relies on the inference result from the previous model and we next describe the inference procedure for these different situations.

2.3.1 Retriever Inference. As aforementioned, ANN indexes are built on the item embeddings for retrievers, which allows for fast and accurate top-k retrieval under numerous items. Given the well-trained item embeddings, we provide the interface of the ANN indexes, including custom index structures and the third-party library, e.g., FAISS[38], where the similarity of item embeddings is encoded. After the user query encoding, the most relevant index centers are selected and the corresponding items are retrieved. In this manner, *Recstudio* achieves efficient inference rather than trivial enumeration over the entire item set.

2.3.2 Cascade Ranker Inference. Under the multi-stage workflow in *RecStudio*, if the ranker is a subsequent model of a retriever of a ranker, the inference process becomes more efficient, where only the top-k ranked items from the preceding model are required to calculate the preference scores. Given the top-k results from the retriever, we first gather the features of items and calculate the preference scores after the interaction layers. These candidate items are then refined depending on the calculated scores to provide more precise top-k results. Rankers commonly involve complex and time-consuming networks, which occupy much time for calculating the preference over the entire item set, accompanied by the huge IO to gather item features. Therefore, it is more efficient to perform a partial computation on only the top-k-ranked items because it takes much less time.

2.3.3 Non-Retrievable Ranker Inference. This situation refers to the independent ranker, where the ranker is trained independently without any preceding input. It is unrealistic for the ranker to score every item in order to determine the preference scores without the previously filtered items. Therefore, the top-k operation is not supported for a single ranker, that is to say, a single ranker could only support rating prediction and CTR tasks, where only the prediction for a pair of the given user and item is required.

2.4 Hyperparameter Setting

2.4.1 Hyperparameter Tuning. Hyperparameters play a significant role in the performance of recommenders and the choice of the appropriate value for hyperparameters is challenging under huge search space. The hyperparameters are tuned depending on the validation dataset whereas the model parameters are learned depending on the training dataset. In *RecStudio*, we can switch between three methods to determine the value of hyperparameters. The first one is **manual setting**, where users are able to run script files with different settings of hyperparameters and manually choose the value with the best performance. The second one is the support of the third-party **automatic tuning** library, i.e., NNI [56], where users may obtain the appropriate hyperparameters after setting the tuning method and tuning range in NNI. The common tuning methods include grid search [42, 43], random search [4], heuristic method [36, 44, 58], bayesian search [18, 35, 45]. In order to further save the time and effort to adjust hyperparameters, we design an automatic learning-based solution assembled within the optimizer.

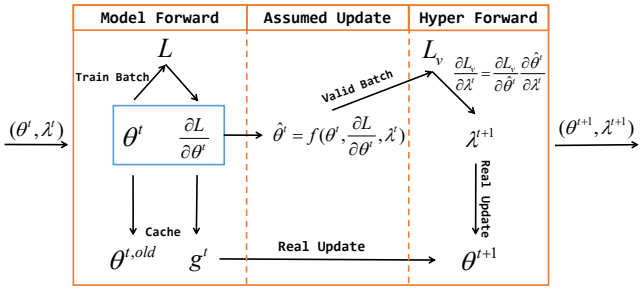

Figure 5: Hyperparemater learning workflow

2.4.2 Hyperparameter Learning. *RecStudio* now features an integrated hyperparameter learning system, which, in addition to updating model parameters, can also update weight decay using learning methods, eliminating the need to manually search for continuous weight decay and allowing for more efficient discovery of its appropriate value. The architecture of the hyperparameter learning system is depicted in Figure 5. Model parameters($\boldsymbol{\theta}$) and weight decay(λ) are alternately learned through the following five stages:

• **Prepare.** Model parameters $\boldsymbol{\theta}^t$ and weight decay λ^t, are calculated from the t^{th} step and inputted for the $t+1^{th}$ step. Initially, $\boldsymbol{\theta}^t$ is randomly initialized and λ^t is set user-specifically.

• **Model Forward.** The training batch is propagated forward at this stage. The associated loss without regularization(L) is obtained. By deriving L to model parameters, we can obtain the corresponding derivatives $\frac{\partial L}{\partial \boldsymbol{\theta}^t}$. At this time, $\boldsymbol{\theta}^{t,old}$ and $\boldsymbol{g}^t$ are duplicated from $\boldsymbol{\theta}^t$ and $\frac{\partial L}{\partial \boldsymbol{\theta}^t}$ and cached for further updates. Since we do not wish to change their values during the subsequent procedure, we cut off all ties between $(\boldsymbol{\theta}^t, \frac{\partial L}{\partial \boldsymbol{\theta}^t})$ and $(\boldsymbol{\theta}^{t,old}, \boldsymbol{g}^t)$. Any operations to $(\boldsymbol{\theta}^t, \frac{\partial L}{\partial \boldsymbol{\theta}^t})$ will not have any effect on $(\boldsymbol{\theta}^{t,old}, \boldsymbol{g}^{t,old})$.

• **Assumed Update.** Assumed model parameter $\hat{\boldsymbol{\theta}}^t$ is calculated by the function $f(\boldsymbol{\theta}^t, \frac{\partial L}{\partial \boldsymbol{\theta}^t}, \lambda^t)$. Under different optimizers, the specific form of f is different. In case of SGD, f has the following form: $\hat{\boldsymbol{\theta}}^t = \boldsymbol{\theta}^t - \eta \frac{\partial L}{\partial \boldsymbol{\theta}^t} - 2\eta\lambda^t\boldsymbol{\theta}^t$, where η represents learning rate.

• **Hyper Forward.** Hyper forward is conducted with the purpose of updating λ^t by its gradient. Valid batch is first input to the model(with parameters as $\hat{\boldsymbol{\theta}}^t$) in order to calculate validation loss L_v. By applying the Chain Rule, derivative of L_v with respect to λ^t could be calculated by $\frac{\partial L_v}{\partial \lambda^t} = \frac{\partial L_v}{\partial \hat{\boldsymbol{\theta}}^t}\frac{\partial \hat{\boldsymbol{\theta}}^t}{\partial \lambda^t}$. Thus, λ^{t+1} can be easily obtained by SGD or Adam optimizers.

• **Real Update.** After λ^{t+1} is obtained, $\boldsymbol{\theta}^{t,old}$ and $\boldsymbol{g}^t$ are loaded into the model. A real update to model parameters could be conducted by a user-specified optimizer. A new version of model parameters $\boldsymbol{\theta}^{t+1}$ and λ^{t+1} are provided for the next iteration.

In Hyper Forward stage, $\frac{\partial L_v}{\partial \hat{\boldsymbol{\theta}}^t}$ and $\frac{\partial \hat{\boldsymbol{\theta}}^t}{\partial \lambda^t}$ are calculated differently. The former could be obtained by the automatic backward function provided by PyTorch. However, the latter should be calculated manually. For example, if SGD is used in the Assumed Update stage, $\frac{\partial \hat{\boldsymbol{\theta}}^t}{\partial \lambda^t}$ equals to $-2\eta\boldsymbol{\theta}^t$ where $\boldsymbol{\theta}^t$ indicates $\boldsymbol{\theta}^{t,old}$ rather than $\hat{\boldsymbol{\theta}}^t$.

To expand the user's flexibility for automatic parameter learning, it remains optional for users to choose whether to use the

Figure 6: Efficient Topk with Mask

```
def topk(self, query, k, user_history):
    # Get top K+|I_train| items
    m = user_history.size(1)
    if self.use_index:
        score, topk_items = self.ann_index.search(query, k+m)
    else:
        scores = self.score_func(query, self.item_vector)
        score, topk_items = torch.topk(scores, k + m)
    # Mask items in I_train with Boolean operation
    existing, _ = user_history.sort()
    idx_ = torch.searchsorted(existing, topk_items)
    idx_[idx_ == existing.size(1)] = existing.size(1) - 1
    mask_ = torch.gather(existing, 1, idx_) == topk_items
    score[mask_] = -torch.inf
    score, idx = score.topk(k)
    topk_items = torch.gather(topk_items, 1, idx)
    return score, topk_items
```

automatic parameter learning module and the update interval of hyperparameters. In future updates of *RecStudio*, we will integrate automatic learning of multiple hyperparameters and develop the learning methods of discrete hyperparameters.

2.5 Evaluation Design

2.5.1 Evaluation Metrics. *RecStudio* equips with a number of evaluation metrics, which cover value-based, CTR-based and ranking-based. The value-based metrics are designed for rating prediction, targeting measuring the difference between the predicted and true ratings, such as Mean Average Error (MAE) and Root Mean Square Error (RMSE). The CTR-based metrics are designed for the CTR task, which models the probability of user clicks. Common metrics used in the task are AUC and logloss. The ranking-based metrics include the widely used ranking-aware metrics, such as Recall, NDCG, Precision, MRR, MAP. Those metrics measure the ranking performance of the recommendation lists. To achieve efficient evaluation, we have implemented GPU-enabled tensor operations and eliminated the dependencies on other libraries.

Another important point is that, different from most existing libraries, which filter the interacted items of training data for the top-k results depending on the set operation, we design and implement a more efficient top-k computation strategy. Specifically, we return actual $K + |I_{train}|$ items, where $|I_{train}|$ denotes the number of interacted items in the training data, and filter the items depending on the Boolean operation. The detailed coding example refers to Figure 6. It is worth noticing that all operations can be conducted on GPUs, without the need to transfer data from GPU to memories for the set computation, achieving efficient and effective top-k retrieval and ensuring only uninteracted items are contained.

2.5.2 Evaluation for Cascading Rankers. As we aforementioned, an independent ranker model can only support rating prediction and CTR prediction, and it is not available to compute the rank-oriented metrics. For cascaded rankers, we can achieve efficient evaluation by only predicting the preference scores over the latest given top-k items. Since the previous model returns the high-ranked items with a wider range, most of the positive samples would be selected in the candidate pool for ranking, and the successive high-precision model is able to rank these positive samples higher, which improves the overall accuracy.

```
# Case 1: Train the model on supported dataset
from recstudio import quickstart  # import the quickstart module in RecStudio
quickstart.run(model='SASRec', dataset='ml-100k', gpu=[2], config=training_config)


# Case 2: Train the model on customized dataset
## Step1: Dataset loading
my_dataset = TripletDataset(name='my_data', config=dataset_config)
## Step2: Spliting the dataset
train,valid,test = my_dataset.build(split_mode='user_entry', split_ratio=[0.8,0.1,0.1])
## Step3: Initialize the model
model = get_model(name='BPR')(config=model_config)
## Step4: Training and Validation
model.fit(train, valid)
## Step5: Evaluation
model.evaluate(test)
```

(a) Running an existing model

```
# Step 1: Build encoders
myQueryEncoder = torch.nn.Embedding(trn.num_users, 64, 0)
myItemEncoder = torch.nn.Embedding(trn.num_items, 64, 0)
# Step 2: Set a score function
myScoreFunc = EuclideanScorer()
# Step 3: Set a negative sampler
mySampler = UniformSampler(trn.num_items)
# Step 4: Set a training loss
myLossFunc = MSELoss()

# Combine all the blocks
myModel = BaseRetriever(
    query_encoder = myQueryEncoder, item_encoder = myItemEncoder,
    scorer = myScoreFunc, loss = myLossFunc, sampler = mySampler
)

# Train and evaluation
myModel.fit(train, valid)
myModel.evaluate(test)
```

(b) Implementing a new model with block building

Figure 7: Code examples for *RecStudio* Usage

3 RECSTUDIO USAGE

In this section, we show how to use *RecStudio* with three code illustrations. We discuss the usage description in three parts: running an existing model, implementing a new model, and employing the web service for the recommendation pipeline.

3.1 Running an Existing Model

3.1.1 Running Models with Specified Hyperparameters. We illustrate the general workflow for running existing models in *RecStudio*, as shown in Figure 7(a). First, one should prepare a dataset configuration for dataset loading and filtering, which should contain dataset download links, names and columns of user/item/interaction files, and other auxiliary parameters. Then some parameters should be provided to split the dataset into train/valid/test. Additionally, the training and evaluation procedure requires some experimental configurations, such as batch size. Note that all the configurations could be obtained by a YAML-format file, a python dict or command line.

3.1.2 Running Models with Auto-tuned Hyperparameters. As mentioned in Section 2.4, *RecStudio* features automatic tuning of hyperparameter for training based on NNI [56]. By specifying the hyperparameters' search space and tuning method as well as NNI's settings like *trialConcurrency* and *maxTrialNumber*, one can quickly find the best solution for a model. The results can be visualized in a web service, which is auto-generated by NNI.

3.2 Implementing a New Model

On top of the modularization of *RecStudio*, one can implement a recommendation model easily by block building or from scratch.

3.2.1 Programming by Building Blocks. In *RecStudio*, Retriever consists of six components: query encoder, item encoder, score function, sampler, loss function, and an optional index structure. One could implement a model by specifying these components, which is just like the game of "building blocks". *RecStudio* is empowered with extensive encoder blocks, such as Multi-Layer Perceptron, Transformer encoder, GRU encoder and CNN encoder. The score function can be selected from Table 1 while the loss function can be picked from Table 2. The valid negative samplers are shown in Table 5. The ANN index can be one of the FAISS[38] indexes or SCANN [25], as long as specified in the configuration file.

Similarly, since the major difference between rankers lies in interaction layers, one can implement a ranker within *RecStudio* in a block-building style, by specifying how to model feature interactions, like cascading explicit interaction layers, followed by some implicit interaction layers. After specifying the loss function, one can include the debiased module for debiasing learning and the contrastive module for contrastive learning.

3.2.2 Programming from Scratch. When implementing a retriever from scratch, the user could inherit the base retriever class by instantiating several functions as follows:

(1) Implementing the "get_dataset_class()" function. In this function, the user is required to assign a dataset class to control the dataset output. Until now, we have implemented four types of dataset classes as shown in Figure 1, namely TripletDataset, UserDataset, SeqDataset, and ALSDataset.
(2) Implementing the "get_query/item_encoder()" function, which builds the encoders as modules.
(3) Implementing the "get_score/loss_func()" function. Those two functions represent the score prediction and training loss calculation as mentioned above.
(4) Implementing the "get_sampler()" function. This function is used to configure the negative sampling method.

When implementing a ranker, the user could inherit the base ranker class, by overriding the explicit feature interaction function "get_interaction_layers()", the implicit interaction function "score()", and the loss function "get_loss_func()".

3.3 A WebService for Pipeline and Benchmark

3.3.1 Build Recommendation Pipeline. Thanks to the modularization design of *RecStudio*, we build a web service for users to build recommendation pipelines like building blocks. Here some steps are listed for detailed usage of the service.

• **Upload a dataset.** User could upload their own dataset by providing configuration, such as dataset name and dataset download link. The dataset is then automatically downloaded for future use.

• **Upload a model.** The model could be also uploaded with a python script file, which should be programmed either by building blocks or from scratch.

• **Build a pipeline.** Once the model and dataset are uploaded, we could use them to establish a recommendation pipeline for training and evaluation. The pipeline for a single model (either retriever or ranker) consists of a dataset and a model. The pipeline of a cascade ranker consists of a dataset, and a pipeline container with retrievers and rankers inside. The supported container includes independent training, ICC [21], RankFlow [57], and CoRR [33], which specify how to connect retrievers and rankers for training and inference. One example of the pipeline is illustrated in Figure 8.

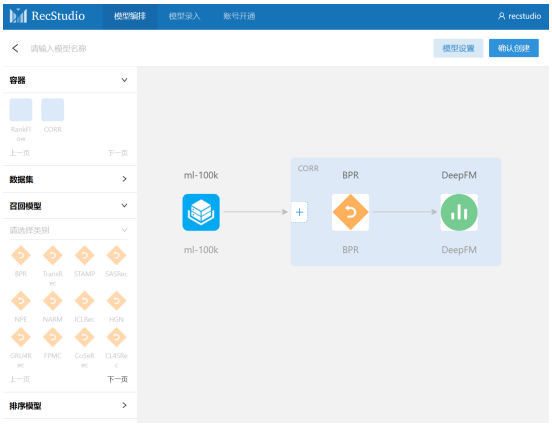

Figure 8: Recommendation pipeline with the web service

• **Training and Logs.** A training job could be submitted once the pipeline is built. Hyperparameter tuning is supported in the job configuration, where users could specify search space and tuning method. Besides, there are several types of logs are provided in the service, like console logs, tensorboard logs. The metrics and loss values can be plotted in real-time when the training goes on.

3.3.2 Automatic Benckmarking. Upon the completion of the training jobs, the training dataset information, the training hyperparameters and the evaluation metrics are automatically sent to a backend database. Therefore, from this database, we establish a database view for archiving the optimal performance of all models on each dataset with each setting. The database view is then connected with a front webservice, which can automatically generate a leaderboard w.r.t a selected dataset with a specified setting. In other words, as long as specifying the dataset name and the settings like split and filtering, any user can observe the performance results of various models in this case and sort them according to any selected metric.

4 INSIGHTS AND DISCUSSIONS

4.1 Comparison with Existing Libraries

With the recent boom in recommender systems, a significant number of open-source libraries for recommender systems have occurred from both industry and academia. We summarize these libraries with plenty of characteristics in Table 6. According to the table, with the popularity of python language and machine learning frameworks, the majority of algorithmic frameworks in recent years have been implemented using python-based frameworks. From the perspective of model type, most recent libraries support deep-learning-based recommenders since 2015 but overlook the traditional yet effective machine-learning-based models. Recstudio implements both traditional algorithms and current popular deep-learning-based models under the PyTorch framework. In addition, Recstudio is equipped with a fast ANN index structure for efficient inference, as well as the GPU-accelerated negative sampling module, which enables high efficiency with massive items.

Recstudio offers an extraordinary user-friendly interaction process, particularly including model building and automatic hyperparameters. On the one hand, the modularized model allows users

Table 6: Comparison with existing recommender system libraries.

Library	Languages	#Models	ModelType	Modularized	HT[a]	ANNs[b]	NS[c]	ReleaseTime
MyMediaLite[22]	C#	61	ML(unspecified)	No	Manual	No	CPU	2010
Crab[9]	Python	2	ML(unspecified)	No	manual	No	CPU	2011
LibFM[60]	C++	1	ML(ranker)	No	manual	No	-	2014
LibRec[23]	Java	93	ML(unspecified)	No	manual	No	CPU	2014
Surprise[34]	Python	11	ML(unspecified)	No	manual	No	CPU	2015
LightFM[40]	Python	1	ML(retriever)	No	manual	No	CPU	2015
Case Recommender[17]	Python	27	ML(unspecified)	No	manual	No	CPU	2015
RankSys[10]	Java	8	ML(unspecified)	No	manual	No	CPU	2016
Spotlight[41]	PyTorch	8	DL(unspecified)	No	tuner	No	CPU	2017
Recommenders[37]	Tensorflow	31	DL(unspecified)	No	tuner	No	CPU	2018
Cornac[65]	Tensorflow	45	DL(unspecified)	No	tuner	No	CPU	2018
DeepCTR[67]	Tensorflow	29	DL(ranker)	Yes	manual	No	-	2018
NeuRec[6]	Tensorflow	33	DL(unspecified)	No	manual	No	CPU	2019
DaisyRec[69]	PyTorch	13	DL(unspecified)	No	tuner	No	CPU	2019
ReChorus[71]	PyTorch	18	DL(unspecified)	No	manual	No	CPU	2020
Beta-recsys[55]	PyTorch	24	DL(unspecified)	No	manual	No	CPU	2020
RecBole[84]	PyTorch	73	DL(unspecified)	No	tuner	No	CPU	2020
TFRS[1]	Tensorflow	2	DL(retriever+ranker)	Partial	manual	Yes	GPU(DNSonly)	2020
Elliot[3]	Tensorflow	50	DL(unspecified)	No	tuner	No	CPU	2021
FuxiCTR[89]	Pytorch	44	DL(ranker)	Yes	tuner	No	-	2021
RecStudio	Pytorch	90	ML/DL(retriever+ranker)	Fully	tuner/learner	Yes	GPU	2022

[a] HT: hyperparameter tuning, [b] ANNs: ANN search indexes, [c] NS: negative sampling

to build models like building blocks without worrying about specific training processes, calculation of evaluation metrics, and other repetitive but tedious steps. In order to implement an algorithm, users only need to determine the recommendation task, determine the data type accordingly, and build each module of the model. Although Recstudio is not the framework that implements the most algorithms, it is possible to quickly implement existing or self-designed models with user-friendly usage. Moreover, Recstudio provides a visual interaction page for newcomers to quickly build their models by dragging and dropping each module on the page and adding configurations. This is especially beneficial for newcomers to get started and attract more interest and research in recommender systems. On the other hand, users are relieved of the need for manual yet time-consuming tuning for hyperparameters. Recstudio supports the automatic setting for hyperparameters, where users are merely required to specify the range of the hyperparameters and the values will be determined automatically by the automatic tuning library or the learning-based solution.

4.2 Insights

Recstudio now supports the modularizing recommendation models, which benefits the convenient model building, and the drag-and-drop programming with Web Service. Furthermore, such modularized Recstudio shows the following insights for step-further research for recommenders:

• **Urgent benchmarks for modules**. Existing benchmarks provide comparisons of the overall algorithm, typically such as the CTR benchmarks [89], but the benchmark for a single module is missing. Plenty of algorithms can be implemented by combining different modules, and if we know the effect of individual models, we may seek for better-performing algorithms.

• **Automatic choices of modules.** These modules now can be assembled like building blocks, leaving the user with the decision of which block to pick. It is intriguing and meaningful to automate the module selection process, which may free up a significant amount of manpower for algorithm design. Each module can simply be viewed as a component in neutral architecture search [90], and intelligent search algorithms can be designed based on the characteristics of the recommendation task.

• **Joint optimization for multi-stage workflow.** The multi-stage workflow tends to produce more accurate recommendation results, especially with cascading rankers [72], but it currently only supports independent learning, where the well-learned models output top-k retrieved items to guide the successive model. This results in a large deviation in distributions for different models, which incurs less accurate results. Therefore, we will integrate various model types for the workflow and perform joint optimization to achieve a better global ranking approximation.

• **Improving few-featured models from enriched information**. A lot of algorithms currently encode users and items only relying on ID information, such as MultiVAE [50] and LightGCN [29], which show superiority in terms of both efficiency and effectiveness. But it is difficult to assemble other information, such as side information and contextual information, into these methods. Improving information utilization for such models is also very interesting, such as a generic model with pre-training modules and a unified framework for distillation from large-scale models.

ACKNOWLEDGMENTS

The work was supported by grants from the National Key R&D Program of China (No. 2021ZD0111801) and the National Natural Science Foundation of China (No. 62022077).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *Osdi*, Vol. 16. Savannah, GA, USA, 265–283.

[2] Qingyao Ai, Vahid Azizi, Xu Chen, and Yongfeng Zhang. 2018. Learning heterogeneous knowledge base embeddings for explainable recommendation. *Algorithms* 11, 9 (2018), 137.

[3] Vito Walter Anelli, Alejandro Bellogín, Antonio Ferrara, Daniele Malitesta, Felice Antonio Merra, Claudio Pomo, Francesco Maria Donini, and Tommaso Di Noia. 2021. Elliot: a comprehensive and rigorous framework for reproducible recommender systems evaluation. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*. 2405–2414.

[4] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).

[5] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*. PMLR, 115–123.

[6] Xiangnan He Xiang Wang Bin Wu, Zhongchuan Sun and Jonathan Staniforth. 2019. *NeuRec*. https://github.com/wubinzzu/NeuRec

[7] Guy Blanc and Steffen Rendle. 2018. Adaptive sampled softmax with kernel based sampling. In *International Conference on Machine Learning*. PMLR, 590–599.

[8] Stephen Bonner and Flavian Vasile. 2018. Causal embeddings for recommendation. In *Proceedings of the 12th ACM conference on recommender systems*. 104–112.

[9] Marcel Caraciolo, Bruno Melo, and Ricardo Caspirro. 2011. Crab: A recommendation engine framework for python. *Jarrodmillman Com* (2011).

[10] Pablo Castells, Neil Hurley, and Saul Vargas. 2021. Novelty and diversity in recommender systems. In *Recommender systems handbook*. Springer, 603–646.

[11] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. 2021. Mongoose: A learnable lsh framework for efficient neural network training. In *International Conference on Learning Representations*.

[12] Chong Chen, Min Zhang, Yongfeng Zhang, Yiqun Liu, and Shaoping Ma. 2020. Efficient neural matrix factorization without sampling for recommendation. *ACM Transactions on Information Systems (TOIS)* 38, 2 (2020), 1–28.

[13] Jin Chen, Defu Lian, Binbin Jin, Xu Huang, Kai Zheng, and Enhong Chen. 2022. Fast variational autoencoder with inverted multi-index for collaborative filtering. In *Proceedings of the ACM Web Conference 2022*. 1944–1954.

[14] Yongjun Chen, Zhiwei Liu, Jia Li, Julian McAuley, and Caiming Xiong. 2022. Intent contrastive learning for sequential recommendation. In *Proceedings of the ACM Web Conference 2022*. 2172–2182.

[15] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[16] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[17] Arthur da Costa, Eduardo Fressato, Fernando Neto, Marcelo Manzato, and Ricardo Campello. 2018. Case recommender: a flexible and extensible python framework for recommender systems. In *Proceedings of the 12th ACM Conference on Recommender Systems*. 494–495.

[18] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*. PMLR, 1437–1446.

[19] Chao Feng, Wuchao Li, Defu Lian, Zheng Liu, and Enhong Chen. 2022. Recommender Forest for Efficient Retrieval. *Advances in Neural Information Processing Systems* 35 (2022), 38912–38924.

[20] Chao Feng, Defu Lian, Zheng Liu, Xing Xie, Le Wu, and Enhong Chen. 2022. Forest-based Deep Recommender. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 523–532.

[21] Luke Gallagher, Ruey-Cheng Chen, Roi Blanco, and J Shane Culpepper. 2019. Joint optimization of cascade ranking models. In *Proceedings of the twelfth ACM international conference on web search and data mining*. 15–23.

[22] Zeno Gantner, Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2011. MyMediaLite: A Free Recommender System Library. In *Proceedings of the 5th ACM Conference on Recommender Systems (RecSys 2011)*.

[23] Guibing Guo, Jie Zhang, Zhu Sun, and Neil Yorke-Smith. 2015. Librec: A java library for recommender systems.. In *UMAP workshops*, Vol. 4. Citeseer, 38–45.

[24] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a factorization-machine based neural network for CTR prediction. (2017), 1725–1731.

[25] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *International Conference on Machine Learning*.

[26] Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J Smola. 2022. BLISS: A Billion scale Index using Iterative Re-partitioning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 486–495.

[27] Ruining He, Wang-Cheng Kang, and Julian McAuley. 2017. Translation-based recommendation. In *Proceedings of the eleventh ACM conference on recommender systems*. 161–169.

[28] Xiangnan He and Tat-Seng Chua. 2017. Neural factorization machines for sparse predictive analytics. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*. 355–364.

[29] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 639–648.

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

[31] Cheng-Kang Hsieh, Longqi Yang, Yin Cui, Tsung-Yi Lin, Serge Belongie, and Deborah Estrin. 2017. Collaborative metric learning. In *Proceedings of the 26th international conference on world wide web*. 193–201.

[32] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE international conference on data mining*. Ieee, 263–272.

[33] Xu Huang, Defu Lian, Jin Chen, Zheng Liu, Xing Xie, and Enhong Chen. 2022. Cooperative Retriever and Ranker in Deep Recommenders. *arXiv preprint arXiv:2206.14649* (2022).

[34] Nicolas Hug. 2020. Surprise: A Python library for recommender systems. *Journal of Open Source Software* 5, 52 (2020), 2174. https://doi.org/10.21105/joss.02174

[35] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*. Springer, 507–523.

[36] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).

[37] Dietmar Jannach, Pearl Pu, Francesco Ricci, and Markus Zanker. 2022. Recommender systems: Trends and frontiers. , 145–150 pages.

[38] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[39] Wang-Cheng Kang and Julian McAuley. 2018. Self-attentive sequential recommendation. In *2018 IEEE international conference on data mining (ICDM)*. IEEE, 197–206.

[40] Maciej Kula. 2015. Metadata Embeddings for User and Item Cold-start Recommendations. In *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.*, Toine Bogers and Marijn Koolen (Eds.), Vol. 1448. 14–21.

[41] Maciej Kula. 2017. Spotlight. https://github.com/maciejkula/spotlight.

[42] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*. 473–480.

[43] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. 2002. Efficient backprop. In *Neural networks: Tricks of the trade*. Springer, 9–50.

[44] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.

[45] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. 2018. Metis: Robustly tuning tail latencies of cloud systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 981–992.

[46] Defu Lian, Qi Liu, and Enhong Chen. 2020. Personalized ranking with importance sampling. In *Proceedings of The Web Conference 2020*. 1093–1103.

[47] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1754–1763.

[48] Dawen Liang, Laurent Charlin, and David M Blei. 2016. Causal inference for recommendation. In *Causation: Foundation to Application, Workshop at UAI. AUAI*.

[49] Dawen Liang, Laurent Charlin, James McInerney, and David M Blei. 2016. Modeling user exposure in recommendation. In *Proceedings of the 25th international conference on World Wide Web*. 951–961.

[50] Dawen Liang, Rahul G Krishnan, Matthew D Hoffman, and Tony Jebara. 2018. Variational autoencoders for collaborative filtering. In *Proceedings of the 2018 world wide web conference*. 689–698.

[51] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv preprint arXiv:1807.05118* (2018).

[52] Zihan Lin, Changxin Tian, Yupeng Hou, and Wayne Xin Zhao. 2022. Improving graph collaborative filtering with neighborhood-enriched contrastive learning.

In *Proceedings of the ACM Web Conference 2022*. 2320–2329.
[53] Zhiwei Liu, Yongjun Chen, Jia Li, Philip S Yu, Julian McAuley, and Caiming Xiong. 2021. Contrastive self-supervised sequential recommendation with robust augmentation. *arXiv preprint arXiv:2108.06479* (2021).
[54] Kelong Mao, Jieming Zhu, Jinpeng Wang, Quanyu Dai, Zhenhua Dong, Xi Xiao, and Xiuqiang He. 2021. SimpleX: A simple and strong baseline for collaborative filtering. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1243–1252.
[55] Zaiqiao Meng, Richard McCreadie, Craig Macdonald, Iadh Ounis, Siwei Liu, Yaxiong Wu, Xi Wang, Shangsong Liang, Yucheng Liang, Guangtao Zeng, et al. 2020. BETA-Rec: Build, Evaluate and Tune Automated Recommender Systems. In *Fourteenth ACM Conference on Recommender Systems*. 588–590.
[56] Microsoft. 2021. *Neural Network Intelligence*. https://github.com/microsoft/nni
[57] Jiarui Qin, Jiachen Zhu, Bo Chen, Zhirong Liu, Weiwen Liu, Ruiming Tang, Rui Zhang, Yong Yu, and Weinan Zhang. 2022. RankFlow: Joint Optimization of Multi-Stage Cascade Ranking Systems as Flows. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 814–824.
[58] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*. PMLR, 2902–2911.
[59] Steffen Rendle. 2010. Factorization machines. In *2010 IEEE International conference on data mining*. IEEE, 995–1000.
[60] Steffen Rendle. 2012. Factorization Machines with libFM. *ACM Trans. Intell. Syst. Technol.* 3, 3, Article 57 (May 2012), 22 pages.
[61] Steffen Rendle and Christoph Freudenthaler. 2014. Improving pairwise learning for item recommendation from implicit feedback. In *Proceedings of the 7th ACM international conference on Web search and data mining*. 273–282.
[62] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. (2009), 452–461.
[63] Yuta Saito. 2020. Unbiased pairwise learning from biased implicit feedback. In *Proceedings of the 2020 ACM SIGIR on International Conference on Theory of Information Retrieval*. 5–12.
[64] Yuta Saito, Suguru Yaginuma, Yuta Nishino, Hayato Sakata, and Kazuhide Nakata. 2020. Unbiased recommender learning from missing-not-at-random implicit feedback. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 501–509.
[65] Aghiles Salah, Quoc-Tuan Truong, and Hady W Lauw. 2020. Cornac: A Comparative Framework for Multimodal Recommender Systems. *Journal of Machine Learning Research* 21, 95 (2020), 1–5.
[66] Tobias Schnabel, Adith Swaminathan, Ashudeep Singh, Navin Chandak, and Thorsten Joachims. 2016. Recommendations as treatments: Debiasing learning and evaluation. In *international conference on machine learning*. PMLR, 1670–1679.
[67] Weichen Shen. 2017. DeepCTR: Easy-to-use,Modular and Extendible package of deep-learning based CTR models. https://github.com/shenweichen/deepctr.
[68] Ryan Spring and Anshumali Shrivastava. 2017. A new unbiased and efficient class of lsh-based samplers and estimators for partition function computation in log-linear models. *arXiv preprint arXiv:1703.05160* (2017).
[69] Zhu Sun, Hui Fang, Jie Yang, Xinghua Qu, Hongyang Liu, Di Yu, Yew-Soon Ong, and Jie Zhang. 2022. DaisyRec 2.0: Benchmarking Recommendation for Rigorous Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).
[70] Scikit-Optimize Team. 2016. Scikit-Optimize. https://scikit-optimize.github.io/stable/
[71] Chenyang Wang, Min Zhang, Weizhi Ma, Yiqun Liu, and Shaoping Ma. 2020. Make it a chorus: knowledge-and time-aware item modeling for sequential recommendation. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 109–118.
[72] Lidan Wang, Jimmy Lin, and Donald Metzler. 2011. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. 105–114.
[73] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. 1–7.
[74] Tianxin Wei, Fuli Feng, Jiawei Chen, Ziwei Wu, Jinfeng Yi, and Xiangnan He. 2021. Model-agnostic counterfactual reasoning for eliminating popularity bias in recommender system. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1791–1800.
[75] Jason Weston, Samy Bengio, and Nicolas Usunier. 2011. Wsabie: Scaling up to large vocabulary image annotation. (2011), 2764–2770.
[76] Jiancan Wu, Xiang Wang, Fuli Feng, Xiangnan He, Liang Chen, Jianxun Lian, and Xing Xie. 2021. Self-supervised graph learning for recommendation. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*. 726–735.
[77] Xu Xie, Fei Sun, Zhaoyang Liu, Shiwen Wu, Jinyang Gao, Jiandong Zhang, Bolin Ding, and Bin Cui. 2022. Contrastive learning for sequential recommendation. In *2022 IEEE 38th international conference on data engineering (ICDE)*. IEEE, 1259–1273.
[78] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed Chi. 2019. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 269–277.
[79] Junliang Yu, Hongzhi Yin, Xin Xia, Tong Chen, Lizhen Cui, and Quoc Viet Hung Nguyen. 2022. Are graph augmentations necessary? simple graph contrastive learning for recommendation. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1294–1303.
[80] Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. 2016. Collaborative knowledge base embedding for recommender systems. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 353–362.
[81] Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. 2016. Collaborative knowledge base embedding for recommender systems. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 353–362.
[82] Weinan Zhang, Tianqi Chen, Jun Wang, and Yong Yu. 2013. Optimizing top-n collaborative filtering via dynamic negative item sampling. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. 785–788.
[83] Yang Zhang, Fuli Feng, Xiangnan He, Tianxin Wei, Chonggang Song, Guohui Ling, and Yongdong Zhang. 2021. Causal intervention for leveraging popularity bias in recommendation. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 11–20.
[84] Wayne Xin Zhao, Yupeng Hou, Xingyu Pan, Chen Yang, Zeyu Zhang, Zihan Lin, Jingsen Zhang, Shuqing Bian, Jiakai Tang, Wenqi Sun, et al. 2022. RecBole 2.0: Towards a More Up-to-Date Recommendation Library. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 4722–4726.
[85] Wayne Xin Zhao, Shanlei Mu, Yupeng Hou, Zihan Lin, Yushuo Chen, Xingyu Pan, Kaiyuan Li, Yujie Lu, Hui Wang, Changxin Tian, et al. 2021. Recbole: Towards a unified, comprehensive and efficient framework for recommendation algorithms. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 4653–4664.
[86] Yu Zheng, Chen Gao, Xiang Li, Xiangnan He, Yong Li, and Depeng Jin. 2021. Disentangling user interest and conformity for recommendation with causal embedding. In *Proceedings of the Web Conference 2021*. 2980–2991.
[87] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1059–1068.
[88] 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 *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1079–1088.
[89] Jieming Zhu, Jinyang Liu, Shuai Yang, Qi Zhang, and Xiuqiang He. 2021. Open benchmarking for click-through rate prediction. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2759–2769.
[90] Barret Zoph and Quoc Le. 2017. Neural Architecture Search with Reinforcement Learning. (2017).