HPT4Rec: AutoML-based Hyperparameter Self-Tuning Framework for Session-based Recommender Systems

Amir Reza Mohammadi[1], Amir Hossein Karimi[2], Mahdi Bohlouli[3], Eva Zangerle[1] and Günther Specht[1]

[1]Department of Computer Science, Universität Innsbruck, Austria
[2]Mathematics and Computer Science Department, Amirkabir University of Technology, Tehran, Iran
[3]Computer Science and Information Technology Department, IASBS, Zanjan, Iran

Abstract

Recommender systems have evolved beyond the basic user-item filtering methods in research. However, these filtering methods are still commonly used in real-world scenarios, mainly because they are easier to debug and reconfigure. Indeed the existing frameworks do not adequately support algorithmic tuning. Moreover, they are primarily focused on the reproducibility of state-of-the-art accuracy rather than ease of algorithm development and maintenance. Therefore, rapid and iterative experimentation and debugging are considerably hindered. In this work, we propose an AutoML-based framework with a modular deep session-based recommender code-base and an integrated automated **H**yper**P**arameter **T**uning (HPT4Rec) component. The proposed framework automates searching for the best session-based model for a given data. Therefore it can help to consistently update the model based on potential changes in the type and volume of data that is prevalent for a real-world scenario. It is demonstrated that HPT4Rec provides extensible data structures, training service compatibility, and GPU-accelerated execution while maintaining training efficiency and recommendation accuracy. We have conducted our experiments on the benchmark RecSys 2015 dataset and achieved performance on par with state-of-the-art results. Achieved results of our experiments show the importance of continuous and iterative parameter tuning, particularly for real-world scenarios.

Keywords

AutoML, Session-based Recommender Systems, Framework, Hyperparameter Tuning

1. Introduction

It is often overwhelming to an e-commerce user to see so many products available for sale. Recognizing the burden of data overload, recommender systems (RSs) improve user experience substantially in various applications. Traditional RSs often rely on user profiles to provide personalized recommendations. Collaborative filtering approaches [1, 2, 3] could use history of purchases to determine user similarity, or use matrix factorization to establish latent factor vectors for each user. In both cases, it is essential to identify the user when making recommendations. However, this may not always be possible, such as not being logged in, having deleted their tracking information, or a new user not having profile. Consequently, recommendation methods that require the user's history suffer from cold-start issues.

Making session-based recommendations is another alternative to using historical data [4]. In this setup, recommendations are only made based on the behavior of users in their current session which helps on tackling the cold-start problem. Session-based recommendation might be a vital component of the future recommendation, especially for the business and real-world applications, as there are concerns and regulations about collecting user data like GDPR [5].

Methods based on deep learning (DL) have shown great promise in the session-based recommendation and also in other communities [6]. As stated in various literature [7, 8, 9], they perform better than traditional baseline methods by around 20-30 percent. However, recent investigations have shown that many of these methods are not compelling enough [10], moreover, results are hard to reproduce in many of them [11], and the codes are not readily available. Recent publications have addressed reproducibility by implementing several DL-based recommendation algorithms as a framework [12, 13, 14]. While these frameworks are effective and helped to alleviate the problem, two key factors should not be overlooked: 1. *Iterative* algorithm optimization: If these algorithms are intended for real-world use, they should include tools for being iteratively tuned to a given dataset (not the offline benchmark datasets). The process should be iterative and persistent since new features may emerge, and user preferences may change. 2. Modularity and ease of reproduciblity: Besides accuracy, several other factors must be taken into consideration, when im-

GVDB23: 34th workshop on basics of database systems (Grundlagen von Datenbanken), June 07–9, 2023, Baden-Württemberg, Germany
amir.reza@uibk.ac.at (A. R. Mohammadi); ahkarimi@aut.ac.ir (A. H. Karimi)
0000-0003-3934-6941 (A. R. Mohammadi); 0009-0001-3946-6954 (A. H. Karimi)

plementing literature-approved methods in production, including non-complexity, fault tolerance, real-time prediction, debuggability, resource consumption, and modularity [15, 16]. The most advanced and well-performing models are often left behind in the business, because they are complex and challenging to debug. As a result, businesses still opt for more straightforward methods that are less accurate, but easier to manipulate and debug. In several papers [8, 10, 17, 18] (discussed in the background section of prior work), various techniques were used to slightly improve performance, which not only may not be useful for large-scale day-to-day use, but may also cause problems in production and during debugging. It would be more practical to implement a robust and modular core structure with clear interfaces and to give room to add more complex mechanisms based on the business demands.

Motivated by reasons mentioned above, in this paper, we present HPT4Rec, an AutoML-based framework for hyperparameter self-tuning with a modular code-base aimed at session-based recommendation. Our framework simplifies the development and manipulation of deep recommendation algorithms to meet business needs. PyTorch and Microsoft NNI [1] are used to develop the codebase, both of which are well known in the DL and AutoML communities and receive continuous updates.

Besides being open-source, this framework can be installed easily, and all prepared data and trained models are available at https://github.com/amirreza-m95/HPT4Rec

2. Prior Work

Background. The most commonly used deep model, when dealing with sequential data are Recurrent Neural Networks (RNN). There is a type of RNNs known as LSTM [19] that are shown to work particularly well, including additional gates regulating, when to take into account input and, when to reset the hidden state. These models are not affected by the vanishing gradient problem usually associated with RNN models. A somewhat simpler alternative to LSTM, but still retaining all of its properties, are Gated Recurrent Units (GRUs) [20], which we employ in this work as the core learning structure of the recommender for the experiments.

Hidasi et al. [7] suggested the RNN approach for session-based recommendation (SBR) and then proposed a parallel RNN architecture [9] to model sessions using the clicks and features of the clicked items. Further research was presented based on RNN methods in order to improve the accuracy of this model. Performance of the recurrent model can be boosted by taking into account temporal changes in user behavior and data augmentation techniques[8]. By uniting the recurrent method with the neighborhood-based method, Jannach et al. [10] combined sequential patterns and co-occurrence signals to get the best of both worlds. Tuan et al. [17] fused session clicks with content features (namely, item titles and categories) to generate recommendations based on 3-dimensional Convolutional Neural Networks (CNN). Li et al. [21] have developed a neural attentive recommendation machine (NARM) using an encoder-decoder architecture. NARM can distinguish sequential behavior and the primary purposes of users using the attention mechanism on RNN. In another study, a Short-Term Attention Priority model (STAMP) [18], which employs a simple MLP network, and an attentive net has been proposed for understanding users' general interests as well as their current interests. In both NARM and STAMP, an attention mechanism emphasizes the importance of the last click.

Almost all of the aforementioned RNN-based SBR models follow the same architecture as GRU4Rec [7]. They have just incorporated new features and mechanisms to improve performance on top of the core structure. Therefore, in HPT4Rec, a minimal code-base based on GRU4Rec was built, with all the necessary tools and modules for a methodologically simplified bottom-up approach to model development. This can remove the barrier of entry for practitioners and allow them to add other features if necessary.

Related Frameworks In the modern RSs field, reproducibility is crucial. Recently, various researchers [10, 11, 22, 23] pointed out the need for fair evaluation of recommender models. Upon thorough hyperparameter tuning, their argument about the supremacy of latent-factor models over deep neural models made it necessary to develop new recommendation frameworks. Beginning in 2011, Mymedialite [24], , RankSys [25], LensKit [26], LightFM [27], and Surprise [28] have established a set of integrated tools for rapid prototyping and testing of recommendation models, using standard metrics and an intuitive model execution. Deep learning (DL) recommendation models achieved remarkable success and attracted growing community interest, which led to the development of new tools. The first open-source frameworks for DL-based recommenders were LibRec [29], Spotlight [30], and OpenRec [31]. Although these frameworks provided plenty of models, they lacked filtering and *Automated* hyperparameter tuning strategies. The RecQ [32], DeepRec [33], and Cornac [34] frameworks have made a significant contribution towards a more comprehensive collection of model implementations. DaisyRec [35], RecBole [36], and Elliot [12] raised the bar considerably after the reproducibility hype, making available a large number of models, data filtering and splitting operations, as well as hyperparameter tuning. Nevertheless, we observed a

[1] https://github.com/microsoft/nni

deficiency of two increasingly critical aspects of recommendation model development in real-world scenarios: *Automated* Hyperparameter tuning and industry-level compatibility of tools and training services. In reviewing these related frameworks, we observed the lack of an open-source recommendation framework to perform automated hyperparameter tuning while adopting various hyperparameter tuning strategies on different distributed platforms. HPT4Rec represents a step toward reaching that goal.

Earlier studies attempted to find a universal automated solution for both architecture design [37, 38] and optimization [39, 40, 41] but that seems to be ineffective since the problems are diverse with different characteristics, so a one-size-fits-all solution is not appropriate. The goal of complete automation might be inspiring for scientific research and serve as a long-term engineering objective, but it seems likely that we will need to semi-automate the majority of these tasks and gradually reduce the human factor over time. Then it is expected that we will develop powerful tools to assist in making machine learning, first and foremost, more systematic and second, more efficient. Aiming to accomplish this goal is the purpose of HPT4Rec.

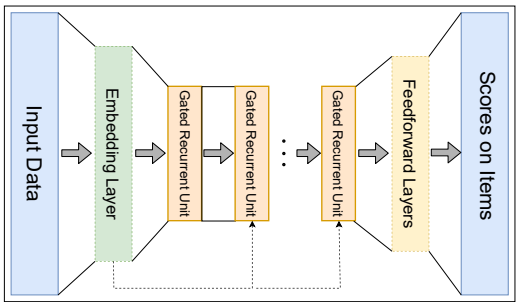

Figure 1: Overview of HPT4Rec's Session-based Recommendation Architecture

3. HPT4Rec

In this section, we describe HPT4Rec's architecture and tuning pipeline. First, we describe the general architecture of the recommender. Next, we present the components and architecture of the framework. Finally, we discuss the available self-tuning methods and their best application scenarios.

3.1. Sequential Modeling with RNN

Variable-length sequence data can be modeled using RNNs. RNNs are characterized by the internal hidden state present in the units that make up the network, which sets them apart from conventional feedforward neural networks. A standard RNN updates its hidden state h according to mechanism showed in eq. (1):

$$\mathbf{h_t} = g\left(W\mathbf{x_t} + U\mathbf{h_{t-1}}\right) \tag{1}$$

where, The logistic sigmoid function g is a smooth function with a bounded input of x_t, which is the unit input at time t. Based on its actual state h_t, an RNN provides a probability distribution for the subsequent element of the sequence.

GRU is a form of RNN that tends to cope with vanishing gradient problems better than vanilla RNN. In essence, GRU gates learn when to update their hidden state and by how much. GRUs are superior to Long Short-Term Memory (LSTM) units when it comes to the session-based recommendation. [7].

A linear interpolation between the prior activation and the candidate activation is used to determine GRU activation, h_t:

$$\mathbf{h_t} = \left(1 - \mathbf{z_t}\right)\mathbf{h_{t-1}} + \mathbf{z_t}\hat{\mathbf{h_t}} \tag{2}$$

where the update gate is given by:

$$\mathbf{z_t} = \sigma\left(W_z\mathbf{x_t} + U_z\mathbf{h_{t-1}}\right) \tag{3}$$

In a similar manner while the candidate activation function, h_t, is also computed:

$$\hat{\mathbf{h_t}} = \tanh\left(W\mathbf{x_t} + U\left(\mathbf{r_t} \odot \mathbf{h_{t-1}}\right)\right) \tag{4}$$

and eventually, the reset gate r_t is provided by:

$$\mathbf{r_t} = \sigma\left(W_r\mathbf{x_t} + U_r\mathbf{h_{t-1}}\right) \tag{5}$$

We have presented the standard formulation of GRU in Equations (3) and (4), but it is important to note that framework users can tweak the model by using other options, like using different final activations such as relu, leaky-relu, and softmax.

3.1.1. GRU4Rec Architecture

The network core comprises the GRU layers, and further feedforward layers may be added between the GRU layer and the output. Each item's predicted preference can be calculated to predict whether it will be the next item in the session. If more than one GRU layer is employed, the hidden state of each layer is used as an input for the next layer. An option is to connect the input to a higher layer of the network to improve performance [7]. We adjusted the base network to suit the task better since recommender systems are not the principal application area of RNNs. The SBR model architecture is demonstrated in Figure 1.

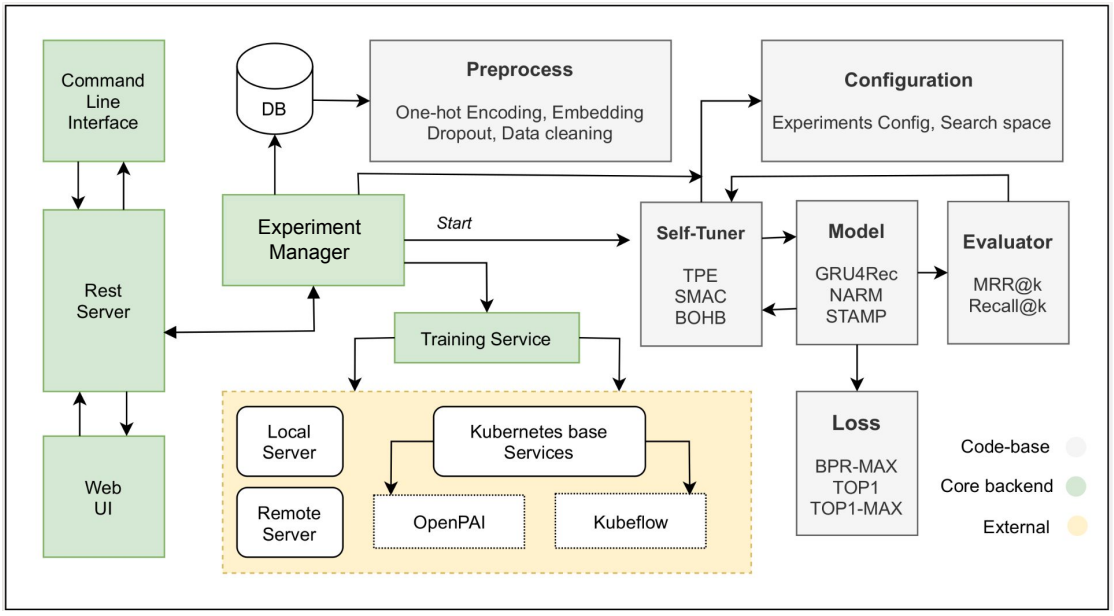

Figure 2: HPT4Rec's Architecture Overview

In addition, we also use trainable embeddings to represent all of our inputs. With backpropagation Through-Time (BPTT), we can train our neural networks using mini-batch gradient descent on multiple options for loss over a dynamic number of time steps.

Session-parallel mini-batches. Click sessions are often of varying length. It may take some users a long time to find their desired item, while others find it within seconds. In the recommender system, accurate predictions should be provided regardless of the current session length. This problem has been addressed by different methods like session-parallel mini-batches [9] and data augmentation [8]. Since we are seeking the least sophisticated approach, we have taken the former approach.

3.2. Architecture and Data Flow

Automated tuning of hyperparameters is a key feature of HPT4Rec. We provide 11 popular self-tuning algorithms. Experiments can be run on a wide range of training platforms, including local machines, multiple servers on a distributed network, and open-source platforms such as Kubernetes and OpenPAI.

3.2.1. HPT4Rec's Data Flow

HPT4Rec experiments are individual attempts to apply a configuration (e.g., a set of hyperparameters) to a model. The first step in constructing an experiment is to define the search space (i.e., parameters). The tuner will sample parameters/architecture according to the search space, which is defined as a JSON file. Search spaces are defined by the variable name, sampling strategy, and parameters of a search space.

A search space definition can be expressed as follows:

```
{
  "dropout_rate": {"_type": "uniform", "_value": [0.1, 0.5]},
  "conv_size": {"_type": "choice", "_value": [2, 3, 5, 7]},
  "hidden_size": {"_type": "choice", "_value": [124, 512, 1024]},
  "lr": {"_type": "loguniform", "_value": [0.0001, 0.1]},
  "momentum": {"_type": "lognormal", "_value": [0.1, 1]}
}
```

We have five parameters to tune in this search space. According to this definition, the dropout rate is characterized by a uniform distribution within a range of 0.1 to 0.5. This search space will be used by Tuner to build configurations, selecting a value from within the range for each parameter. Besides defining the search space, the only requirement is to define a configuration file containing information like experiment log folder, self-tuning algorithms, trial number, and duration threshold. The configuration file is in YAML format.

In order to implement a new tuning algorithm or tweak the existing ones, the base tuner should be inherited. Then, by following the interface of the module and returning the experiment results, passing the new parameters, and updating the search space, the tuning module will function properly.

Table 1
Self-tuning methods performance on different proxy datasets.

	TPE			SMAC			Anneal		
#Samples	Recall@20	MRR@20	Time	Recall@20	MRR@20	Time	Recall@20	MRR@20	Time
125K	0.4314	0.2069	**23**	0.4229	**0.2114**	29	**0.4332**	0.203	25
250K	0.4687	0.225	**39**	**0.473**	0.2235	45	0.4633	**0.2311**	41
500K	0.5062	0.2426	76	0.5082	0.2442	77	**0.5103**	**0.2487**	**57**
1M	0.545	0.2559	**139**	0.5479	**0.2636**	147	**0.5481**	0.2619	191

3.2.2. Architecture

By executing the experiment_runner python script through Cli and passing the configuration file path, experiments are instantiated. The experiment manager parses the configuration file to determine the path to the search space and target the training service, and then runs the model code with the appropriate parameters from the search space. Preprocessing will be performed by the experiment manager (e.g., one-hot encoding, embedding dropout). Following the execution of the model with the first set of parameters, the self-tuner will examine intermediate results (i.e., after each epoch) to determine whether results are improving. Next, it will pass the model on to the evaluation module. Evaluation will be conducted by the evaluator, and results will be provided to self-tuning algorithm to update its inner state. Following the update, the self-tuning algorithm determines the next metric to use. The iterative process will be repeated until a certain time or number of experiments is reached. Figure 2 illustrates this procedure. HPT4Rec will output results in a webUI interface and collect all metrics, intermediate results, best parameters, and system logs in a JSON format.

3.2.3. Self-tuning

The cycle of getting hyperparameters, carrying out experiments, testing their results, and then tuning hyperparameters is deemed as self-tuning. Recommender systems are used in various online websites with different levels of user activity, which directly affects the volume of data available for training models. Additionally, Training deep models require substantial computational resources, which is another crucial aspect since it directly impacts revenue. Thereby, different tuning strategies are needed based on available features, the volume of data, and available computational resources. Based on the framework review shown in table 1, HPT4Rec offers several tuning techniques tailored for diverse scenarios that occur in real-world scenarios.

After a series of experiments, we have gained an early intuition about the most suitable use cases of each self-tuning algorithm. In that sense, Tree-structured Parzen Estimator (TPE) [42] is suitable when computation resources are limited, and you can only try a limited number of trials. A wide range of experiments revealed that TPE outperformed random search. If the variables in the search space can be selected from a prior distribution, Anneal is useful. Likewise, it is recommended to use naive evolution, when your experiment code supports weight transfer, which implies that the experiment could inherit its parent's converged weight from its predecessor. Training can be substantially accelerated with the right tuning method, resulting in less time and money spent and higher revenue, as well as better recommenders, to enhance user experience.

4. Experiments

4.1. Experiment Setup

4.1.1. Dataset

We conducted our experiments on the YOOCHOOSE e-commerce dataset for RecSys 2015 challenge [2]. A six-month period of click-streams from an e-commerce site was included in this dataset. Click-streams are sometimes followed by purchase events. Following preprocessing, there are 7,936,469 sessions and 31,437,691 clicks on 37,403 items left for training and testing. Each clicking event contains a session ID, an item ID and, if the item is a buy-item, a price tag. A shopping session can contain anywhere between 1 and 200 clicks, but most sessions contain less than 30 clicks. We keep only the click events from the challenge's training set. Sessions of length one are filtered out. The Yoochoose dataset was chosen since it is the most general dataset, based on the dataset's features compared to other well-known datasets in this field such as Diginetica[3], Xing[4], and Last.fm[5]. The default settings of the framework can be used for all the datasets we mentioned just by omitting some of their extra features. We employ a dataset characterized by minimalistic data features as a means to ensure the robust generalizability of the model to diverse datasets encompassing a greater abundance of data features.

[2] http://2015.recsyschallenge.com
[3] https://competitions.codalab.org/competitions/11161
[4] http://2016.recsyschallenge.com/
[5] http://ocelma.net/MusicRecommendationDataset/lastfm-1K.html

4.1.2. Evaluation Metrics

In order to match the user with the most relevant item on the list, recommender systems can recommend only a few items at a time. We, therefore, use recall@20 as our main evaluation metric, which counts the proportion of cases that have the targeted item in the top 20 items for all test cases. As long as an item is among the top-N, recall does not take its rank into consideration. The MRR@20 metric is the second metric used in the experiments. A reciprocal ranking of the desired items determines this value. A reciprocal rank above 20 is set to zero.

4.1.3. Implementation Details

For demonstration purposes and to have a quantifiable search space, we optimized hidden size, batch size, learning rate and the number of GRU layers and fixed others as follow. For our model, 50-dimensional embeddings were used for the items, with a 20% embedding dropout. The optimization was conducted using Adam [43]. The GRU search space was set at 50 to 1000 hidden units for each model. A session ends with the GRU's hidden state reset to zero. Models are developed in PyTorch and trained on an NVIDIA Tesla V100. The source code of the model, checkpoints, and logs are available online.

The comparison was made with four traditional recommendations (POP, S-POP, Item-KNN and BPR-MF) and with two well-performing configurations of GRU4Rec.

- *POP.* In one of its simplest forms, the popular predictor predicts the items that are most popular in the training set. Even though it is simple, it often provides a good baseline for certain domains.
- *S-POP.* This baseline recommends the items that are most popular during the current session. As the session progresses, the recommendation list grows. Global popularity values are used to break up ties.
- *Item-KNN.* This baseline measures similarity by dividing the number of times two items appear together in sessions by the square root of the product of their occurrence rates.

4.2. Performance and Results

4.2.1. Diverse Self-tuning Methods Effectiveness

The most likely scenario for developing a recommender system in the real world is carrying out an experiment, where different levels of training data are collected. This may change as user activity increases and new users visit the website. Even in the offline dataset of RecSys 2015, the results of training on a complete dataset are slightly worse than those of training on a recent region of the dataset, which shows changing user behavior [8].

Table 2
Comparison of the our optimized recommender against baselines.

model/type/loss HS [6]	***Recall@20***	***MRR@20***
POP	0.005	0.0012
S-POP	0.2672	0.1775
Item-KNN	0.5065	0.2048
BPR-MF	0.2574	0.0618
GRU4REC BPR 1000	**0.6322**	0.2467
GRU4REC top1 100	0.5853	0.2305
HPT4Rec TOP1 110	0.6259	**0.2681**

Thus, to make recommendations that reflect changes in user behavior over time, models must be continuously and iteratively optimized. It is possible to have different approaches when we have different quantities of data and computation to find the best-optimized model, as discussed in the self-tuning 3.2.2. Our experiments have been conducted using four proxy datasets that mirror the RecSys benchmark data, which comprise different quantities of data. HPT4Rec's recommender model was tuned using four self-tuning methods that used proxy datasets as training data. Evaluation metrics and tuning time were recorded to compare these methods. Table 2 shows how we found the most effective model using 30 experiments. Results do not indicate the optimal use case scenario for tuning methods, but rather demonstrate that each of these tuners performs well in different scenarios and that one of them does not outperform the others in all proxy datasets and evaluation metrics.

4.2.2. Consistency with Published Results

A key element for any new tool is consistency with the previously published results since a wide range of results are possible due to a variety of implementation details, non-fixed seed values, and other domain-specific reasons. Our research also featured HPT4Rec's self-tuning method for optimizing the base recommender model with the Original RecSys dataset. In Table 2 we show that HPT4Rec has outperformed baseline models by a fair margin and is almost on par with state-of-the-art models with this privilege that it has discovered parameters that lead to a simpler model, which results in less resource consumption in production mode. The pursuit of more streamlined models facilitates enhanced reproducibility, a fundamental tenet of our methodology, thereby engendering an essential advancement.

5. Conclusion and Future Work

In this paper, we have released a session-based recommender system framework based on AutoML called HPT4Rec. We reviewed the recommended systems frame-

works in the literature, showing HPT4Rec's merits and shortcomings, and emphasizing the advantages of modularity and automatic tuning. To the best of our knowledge, HPT4Rec is the first recommendation framework that provides a thorough self-tuning experimental pipeline supported by business scale training service compatibility. We expect HPT4Rec to simplify the tuning effort of recommendation models, facilitate the development and debugging process of new algorithms, and help migrate deep recommender algorithms to be used in real-world scenarios. Our immediate future work will emphasize automating other aspects of the recommendation pipeline, such as automated data augmentation, which has traditionally been done manually in literature.

References

[1] Y. Koren, R. Bell, C. Volinsky, Matrix factorization techniques for recommender systems, Computer 42 (2009) 30–37.

[2] Y. Koren, Factorization meets the neighborhood: a multifaceted collaborative filtering model, in: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, 2008, pp. 426–434.

[3] R. Salakhutdinov, A. Mnih, G. Hinton, Restricted boltzmann machines for collaborative filtering, in: Proceedings of the 24th international conference on Machine learning, 2007, pp. 791–798.

[4] J. B. Schafer, J. Konstan, J. Riedl, Recommender systems in e-commerce, in: Proceedings of the 1st ACM conference on Electronic commerce, 1999, pp. 158–166.

[5] E. Commission, 2018 reform of eu data protection rules, 2018-05-25. URL: https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf.

[6] A. Datar, C. Pan, M. Nazeri, X. Xiao, Toward wheeled mobility on vertically challenging terrain: Platforms, datasets, and algorithms, arXiv preprint arXiv:2303.00998 (2023).

[7] B. Hidasi, A. Karatzoglou, L. Baltrunas, D. Tikk, Session-based recommendations with recurrent neural networks, CoRR abs/1511.06939 (2016).

[8] Y. K. Tan, X. Xu, Y. Liu, Improved recurrent neural networks for session-based recommendations, in: Proceedings of the 1st workshop on deep learning for recommender systems, 2016, pp. 17–22.

[9] B. Hidasi, M. Quadrana, A. Karatzoglou, D. Tikk, Parallel recurrent neural network architectures for feature-rich session-based recommendations, in: Proceedings of the 10th ACM conference on recommender systems, 2016, pp. 241–248.

[10] D. Jannach, M. Ludewig, When recurrent neural networks meet the neighborhood for session-based recommendation, in: Proceedings of the Eleventh ACM Conference on Recommender Systems, 2017, pp. 306–310.

[11] M. F. Dacrema, P. Cremonesi, D. Jannach, Are we really making much progress? a worrying analysis of recent neural recommendation approaches, in: Proceedings of the 13th ACM Conference on Recommender Systems, 2019, pp. 101–109.

[12] V. W. Anelli, A. Bellogín, A. Ferrara, D. Malitesta, F. A. Merra, C. Pomo, F. M. Donini, T. D. Noia, Elliot: a comprehensive and rigorous framework for reproducible recommender systems evaluation, 2021. arXiv:2103.02590.

[13] L. Yang, E. Bagdasaryan, J. Gruenstein, C.-K. Hsieh, D. Estrin, Openrec: A modular framework for extensible and adaptable recommendation algorithms, in: Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 664–672. URL: https://doi.org/10.1145/3159652.3159681.

[14] S. Zhang, Y. Tay, L. Yao, B. Wu, A. Sun, Deeprec: An open-source toolkit for deep learning based recommendation, 2019. arXiv:1905.10536.

[15] P. Kouki, I. Fountalis, N. Vasiloglou, X. Cui, E. Liberty, K. Al Jadda, From the lab to production: A case study of session-based recommendations in the home-improvement domain, in: Fourteenth ACM conference on recommender systems, 2020, pp. 140–149.

[16] D. Jannach, M. Jugovac, Measuring the business value of recommender systems, ACM Trans. Manage. Inf. Syst. 10 (2019). URL: https://doi.org/10.1145/3370082.

[17] T. X. Tuan, T. M. Phuong, 3d convolutional networks for session-based recommendation with content features, in: Proceedings of the eleventh ACM conference on recommender systems, 2017, pp. 138–146.

[18] Q. Liu, Y. Zeng, R. Mokhosi, H. Zhang, Stamp: short-term attention/memory priority model for session-based recommendation, in: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018, pp. 1831–1839.

[19] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (1997) 1735–1780.

[20] K. Cho, B. Van Merriënboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: Encoder-decoder approaches, Fifth Workshop on Syntax, Semantics and Structure in Statistical Translation (2014).

[21] J. Li, P. Ren, Z. Chen, Z. Ren, T. Lian, J. Ma, Neural attentive session-based recommendation, in:

Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, 2017, pp. 1419–1428.

[22] S. Rendle, L. Zhang, Y. Koren, On the difficulty of evaluating baselines: A study on recommender systems, 2019. arXiv:1905.01395.

[23] D. Jannach, G. de Souza P. Moreira, E. Oldridge, Why are deep learning models not consistently winning recommender systems competitions yet? a position paper, in: Proceedings of the Recommender Systems Challenge 2020, RecSysChallenge '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 44–49. URL: https://doi.org/10.1145/3415959.3416001.

[24] Z. Gantner, S. Rendle, C. Freudenthaler, L. Schmidt-Thieme, MyMediaLite: A free recommender system library, in: 5th ACM International Conference on Recommender Systems (RecSys 2011), 2011.

[25] S. Vargas, Novelty and diversity enhancement and evaluation in recommender systems and information retrieval, in: Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval, 2014, pp. 1281–1281.

[26] M. D. Ekstrand, Lenskit for python: Next-generation software for recommender systems experiments, in: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, 2020, pp. 2999–3006.

[27] M. Kula, Metadata embeddings for user and item cold-start recommendations, arXiv preprint arXiv:1507.08439 (2015).

[28] N. Hug, Surprise: A python library for recommender systems, Journal of Open Source Software 5 (2020) 2174.

[29] G. Guo, J. Zhang, Z. Sun, N. Yorke-Smith, Librec: A java library for recommender systems., in: UMAP Workshops, volume 4, Citeseer, 2015.

[30] M. Kula, Spotlight, https://github.com/maciejkula/spotlight, 2017.

[31] L. Yang, E. Bagdasaryan, J. Gruenstein, C.-K. Hsieh, D. Estrin, Openrec: A modular framework for extensible and adaptable recommendation algorithms, in: Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, 2018, pp. 664–672.

[32] J. Yu, M. Gao, H. Yin, J. Li, C. Gao, Q. Wang, Generating reliable friends via adversarial training to improve social recommendation, in: 2019 IEEE International Conference on Data Mining (ICDM), IEEE, 2019, pp. 768–777.

[33] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, C.-J. Wu, Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2020, pp. 982–995.

[34] A. Salah, Q.-T. Truong, H. W. Lauw, Cornac: A comparative framework for multimodal recommender systems, Journal of Machine Learning Research 21 (2020) 1–5.

[35] Z. Sun, D. Yu, H. Fang, J. Yang, X. Qu, J. Zhang, C. Geng, Are we evaluating rigorously? benchmarking recommendation for reproducible evaluation and fair comparison, in: Fourteenth ACM Conference on Recommender Systems, RecSys '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 23–32. URL: https://doi.org/10.1145/3383313.3412489.

[36] W. X. Zhao, S. Mu, Y. Hou, Z. Lin, K. Li, Y. Chen, Y. Lu, H. Wang, C. Tian, X. Pan, Y. Min, Z. Feng, X. Fan, X. Chen, P. Wang, W. Ji, Y. Li, X. Wang, J.-R. Wen, Recbole: Towards a unified, comprehensive and efficient framework for recommendation algorithms, 2020. arXiv:2011.01731.

[37] P. Zhao, K. Xiao, Y. Zhang, K. Bian, W. Yan, Amer: Automatic behavior modeling and interaction exploration in recommender system, arXiv preprint arXiv:2006.05933 (2020).

[38] Y. Chen, Y. Yang, H. Sun, Y. Wang, Y. Xu, W. Shen, R. Zhou, Y. Tong, J. Bai, R. Zhang, Autoadr: Automatic model design for ad relevance, in: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, 2020, pp. 2365–2372.

[39] T.-H. Wang, X. Hu, H. Jin, Q. Song, X. Han, Z. Liu, Autorec: An automated recommender system, in: Fourteenth ACM Conference on Recommender Systems, 2020, pp. 582–584.

[40] R. Anand, J. Beel, Auto-surprise: An automated recommender-system (autorecsys) library with tree of parzens estimator (tpe) optimization, in: Fourteenth ACM Conference on Recommender Systems, 2020, pp. 585–587.

[41] H. Liu, X. Zhao, C. Wang, X. Liu, J. Tang, Automated embedding size search in deep recommender systems, in: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, 2020, pp. 2307–2316.

[42] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, in: 25th annual conference on neural information processing systems (NIPS 2011), volume 24, Neural Information Processing Systems Foundation, 2011.

[43] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).