OpenRec: A Modular Framework for Extensible and Adaptable Recommendation Algorithms

Longqi Yang
Cornell Tech, Cornell University
ylongqi@cs.cornell.edu

Eugene Bagdasaryan
Cornell Tech, Cornell University
eugene@cs.cornell.edu

Joshua Gruenstein
Massachusetts Institute of Technology
jgru@mit.edu

Cheng-Kang Hsieh
Cornell Tech, Cornell University
changun@cs.ucla.edu

Deborah Estrin
Cornell Tech, Cornell University
destrin@cornell.edu

ABSTRACT

With the increasing demand for deeper understanding of users' preferences, recommender systems have gone beyond simple user-item filtering and are increasingly sophisticated, comprised of multiple components for analyzing and fusing diverse information. Unfortunately, existing frameworks do not adequately support extensibility and adaptability and consequently pose significant challenges to rapid, iterative, and systematic, experimentation. In this work, we propose **OpenRec**, an open and modular Python framework that supports extensible and adaptable research in recommender systems. Each recommender is modeled as a computational graph that consists of a structured ensemble of reusable modules connected through a set of well-defined interfaces. We present the architecture of OpenRec and demonstrate that OpenRec provides adaptability, modularity and reusability while maintaining training efficiency and recommendation accuracy. Our case study illustrates how OpenRec can support an efficient design process to prototype and benchmark alternative approaches with inter-changeable modules and enable development and evaluation of new algorithms.

KEYWORDS

Recommendation; framework; modular; extensible; adaptable.

ACM Reference format:
Longqi Yang, Eugene Bagdasaryan, Joshua Gruenstein, Cheng-Kang Hsieh, and Deborah Estrin. 2018. OpenRec: A Modular Framework for Extensible and Adaptable Recommendation Algorithms. In *Proceedings of WSDM'18, February 5–9, 2018, Marina Del Rey, CA, USA, ,* 9 pages.
DOI: http://dx.doi.org/10.1145/3159652.3159681

1 INTRODUCTION

Today's recommender systems have gone beyond simple collaborative or content-based filtering algorithms to become large-scale learning machines that ingest and analyze a wide range of information, e.g., diverse user feedback signals (ratings [3], click-through [17], likes [16], views [33]) and auxiliary, contextual and cross-platform traces (images [15], video [8], audio [30] and other associated metadata [25]; as well as social networks [12], software tool traces [33], and personal digital traces [16]). A state-of-the-art system [8] usually involves numerous heterogeneous and complex sub-models that analyze and fuse high-dimensional and multi-channel data streams, and each of these sub-models may have different learning architectures and a large number of hyper-parameters that need to be developed and maintained.

As a result, recommender system developers are facing an exponentially larger design space given the multiple interdependent design decisions that they need to make, such as: (1) which collaborative filtering model to use, (2) which additional data to incorporate, (3) for each additional data, which feature extraction methods to use, and (4) how to integrate the extracted features with the collaborative filtering part of the model. Moreover, researchers' design space now includes: identifying novel data sources to incorporate into the system, developing new feature extractors, and experimenting with new ways to integrate features with the user-item filtering. The software frameworks previously available for recommender systems are limited to "functional level" modularity (e.g., Librec [11] decomposes a recommender system into *inference*, *prediction*, and *similarity*), which does not provide the modularity needed to build and evaluate increasingly complex models. This paper addresses key challenges of **extensibility** and **adaptability**.

On the one hand, traditional frameworks, such as MyMediaLite [10] and LensKit [9], usually treat a recommendation algorithm as single and **monolithic**. As a result, in order to experiment with a new method for even a small part of the algorithm, researchers often need to re-implement the whole model from scratch or extensively patch existing code. For example, to build a recommendation algorithm that incorporates image data, a researcher needs to not only implement the neural network for image analysis, but also re-build the factorization algorithm (e.g. Probablistic Matric Factorization), because there is no interface available in the traditional frameworks to access component modules. Significant rewriting is needed even when the recommendation is a simple composition of existing models.

On the other hand, adapting traditional frameworks to diverse recommendation scenarios requires tedious re-implementations, which may significantly affect recommendation performance despite slight implementation differences (e.g., different choices of hyper-parameters and regularization terms) [9]. We argue that such re-implementations are inevitable if the frameworks are built on diverse backend and programming languages, e.g., Java [11], C# [10], and Python [18], because of the overhead and the opportunity cost

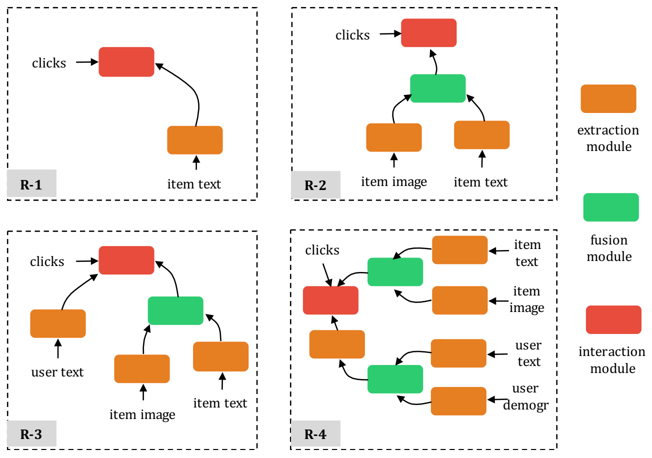

Figure 1: A modular view of recommendation algorithms. Each algorithm (R-1 to R-4) is a structured ensemble of reusable modules under three categories - extraction module, fusion module, and interaction module. The modules' color codex is shared throughout the paper. Arrows in the figure represent data flows.

of switching between different development environments. Additionally, existing frameworks typically assume a single machine environment, which can not leverage the computation power from distributed computing and modern hardware, e.g., GPU and TPU. Therefore, it is hard to be adopted when the model size increases.

To tackle these challenges, we propose a **modular** recommender-system design, where each recommender is a structured ensemble of reusable modules with standard interfaces. This allows the recommendation system innovation to be decomposed into (1) *designing new modules*, and (2) *inventing new computational graphs* that wire modules together. As demonstrated in Fig. 1, future research can readily reuse existing modules and graphs without re-implementing or modifying prior algorithms. Under such a paradigm, changes to a module or the computational graph does not affect other components, and development and testing can be more readily achieved via *plug-in and go*. Just as *modular architectures and tools* lead to rapid advances in other AI fields [5, 7, 24] and network protocol simulation [4], a modular paradigm can significantly reduce the development overhead and become fertile ground for extensible and adaptable recommender system research. In addition, we propose to use Tensorflow [1] as the standard backend for framework development. Because Tensorflow can be easily deployed in diverse computing environments (e.g., embedded devices, single machine, and distributed cloud) and is optimized for modern hardware, its use enables distributed and mini-batch (i.e., large-scale dataset) training for OpenRec and can minimize the need for language-switching re-implementations.

In this work, we present the design, implementation, and evaluation of **OpenRec**, an open and modular framework that supports extensible and adaptable research in recommender systems. Specifically, we build such a framework by (1) modularizing prior recommender systems, (2) identifying reusable modules and defining standard interfaces, and (3) iteratively implementing and developing in Tensorflow. In addition, we evaluate and demonstrate OpenRec in the following three contexts.

- **Reproducing monolithic implementations with OpenRec modular design.** We extensively compare the performance of the modular implementations to the prior implementations and demonstrate that the modularity in OpenRec does not degrade the models' performance in terms of both training efficiency and prediction accuracy. To the contrary, in many cases, OpenRec outperformed the existing implementations due to the ability to conduct large-batch training.
- **Rapid prototyping using OpenRec as a sandbox.** Using book recommendation as an example, we illustrate how developers can use OpenRec to address specific recommendation problems by efficiently prototyping and bench-marking a large number of approaches with modules that are inter-changeable.
- **Developing new recommendation algorithms by extending existing modules in OpenRec.** We use OpenRec to build a time-aware movie rating prediction algorithm for the Netflix dataset. We demonstrate that OpenRec can significantly alleviate the development burden when exploring new techniques.

The OpenRec framework (Apache-2.0) is publicly available under the URL: http://www.openrec.ai

2 EVOLUTION OF RECOMMENDER SYSTEMS

In this section we briefly review the evolution of recommender systems. We discuss how recommender systems have evolved from pure collaborative filtering approaches to hybrid and content-aware models, and discuss the design challenges that arise with such development.

2.1 Pure Collaborative Filtering Models

Early recommender system research focused on designing collaborative filtering models that process users' past user-item interaction data (e.g., ratings, click-through, etc.) to predict what users will like in the future [3, 17]. A well-known example is matrix factorization, where users' past behaviors are encoded in an incomplete user-item matrix, and the prediction is made by estimating the values of the missing cells in the matrix with a low-rank assumption. Matrix factorization and other collaborative filtering models achieved great results in the Netflix competition [3], and a great amount of work has been devoted to improving upon these original approaches. The most recent examples include Neural Collaborative Filtering [14] that utilizes a neural network to allow for non-linear interactions between users and items, and Collaborative Metric Learning [15], that approaches the collaborative filtering problem from a metric learning perspective.

2.2 Hybrid and Content-ware Models

The original use cases of collaborative filtering algorithms were for the scenarios where user-item interactions are abundant (e.g. movie recommendations on Netflix or product recommendations on Amazon [3, 21]) and the user-item interaction data alone is sufficient to make high quality recommendations. However, with digital services becoming more ubiquitous in daily life, there is a increasing demand for recommender systems to work for other scenarios where users have had little or no prior interaction with the system (i.e. the user cold start scenarios), or for the scenarios where candidate items have not received much feedback from users yet (i.e. the item cold start scenarios). Collaborative filtering algorithms work poorly in such scenarios as the amount of interaction data are too sparse for them to reliably estimate users' preferences.

This demand for more powerful and diverse recommender systems, along with the rapid advances in machine learning algorithms for content analysis, has driven a new generation of research that goes beyond the user-item matrix; in particular, new algorithms use various machine learning models to extract relevant features from additional sources [28]. For example, specific algorithms have been designed to extract item features from a large variety of signals, such as text and image data associated with items; similarly, different approaches have been proposed to extract user features from their social media traces, reviews, or other public and personal digital traces [16]. The extracted features are fused with the collaborative filtering portion of the model to allow the system to get a deeper understanding of items and users. Such hybrid models often show superior performance in cold-start scenarios, and continue to outperform the collaborative filtering solutions later on [13, 16]. Moreover, the use of content information also allows for more specific explanations to the recommendation results as compared with the generic "users like you also like this" explanations enabled by prior collaborative filtering based approaches.

3 RELATED FRAMEWORKS

The rapid evolution of recommender systems (Section 2) has posed significant challenges to the existing software frameworks. In this section, we briefly review the limitations of existing solutions, and discuss *why OpenRec is timely* and *is preferable to modularizing existing frameworks*. We show the core functions of previous frameworks and their comparisons to OpenRec in Table. 1. Existing solutions are limited in the following two aspects.

- **Lack of algorithm level modularity support.** Previous frameworks usually provide modularity at the "functional level", i.e., each recommender is divided into functionally-independent components (e.g. train, predict, and dataset). Such functionality-based modularity is convenient while developing new systems, but falls short when it comes to inventing and experimenting with complex algorithms, i.e. developers still need to build algorithms monolithically. In addition, because there is no "algorithm-level modularity", it is non-trivial to add complex auxiliary features into recommendation. Therefore, OpenRec addresses a *timely* need for the recommender system community.
- **Lack of reliable backend support.** As is shown in Table 1, previous frameworks were built on either no explicit backend or a backend that is not scalable and unfriendly to complex models, e.g. Scikit. With such backends, the recommender systems can not leverage modern hardware, such as GPUs, and is hard to scale to distributed computing environment. It is also very cumbersome for the developers to build new functions as there is little support for basic mathematical operations. Therefore, modularizing based on a legacy backend is limiting. We develop OpenRec over Tensorflow, a next generation computing engine for machine learning.

Table 1: Comparing OpenRec to existing software frameworks for recommender systems. (Sys-m: system-level modularity, Algo-m: algorithm-level modularity)

Framework	Sys-m	Auxiliary features	Backend	Algo-m
MLlib [29]	✗	✗	✗	✗
MyMediaLite [10]	✓	categorical	✗	✗
LensKit [9]	✓	✗	✗	✗
Surprise [18]	✓	✗	SciKits	✗
PredictionIO [6]	✓	categorical	✗	✗
Librec [11]	✓	categorical	✗	✗
OpenRec	✓	complex	Tensorflow	✓

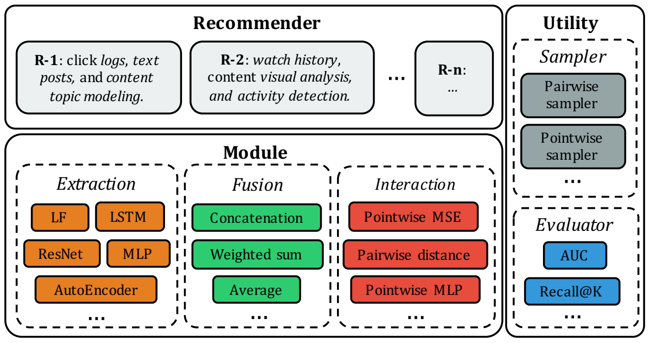

Figure 2: The architecture of OpenRec. A recommender is built out of modules. All three components (Module, Recommender, and Utility) can be used seamlessly together to conduct training, evaluation, experimentation, and serving for recommendation algorithms.

4 OPENREC FRAMEWORK

In this section, we describe the architecture of OpenRec. It views each recommendation algorithm as a computational graph that connects reusable modules together. OpenRec is comprised of two levels of abstractions - **module** and **recommender** - along with a collection of **utility** functions (Fig. 2). Under this framework, a **module** defines standard input/output interfaces for each category of algorithmic component. A **recommender** provides mechanisms to build end-to-end systems out of modules. **Utility** includes functions for efficient data sampling and model evaluation. In the rest of this section, we present the details of each abstraction. Although we illustrate OpenRec with collaborative filtering approaches, the framework is also designed for more general recommendation techniques, e.g., content-based, conversational and group recommendations. We discuss the generalization of the framework in Section 4.4.

4.1 Recommenders

The Recommender abstraction provides a standard way to construct recommendation systems with modules (Section 4.2) and to easily conduct training and testing. The design philosophy behind the recommender is to decouple the construction of a complex system

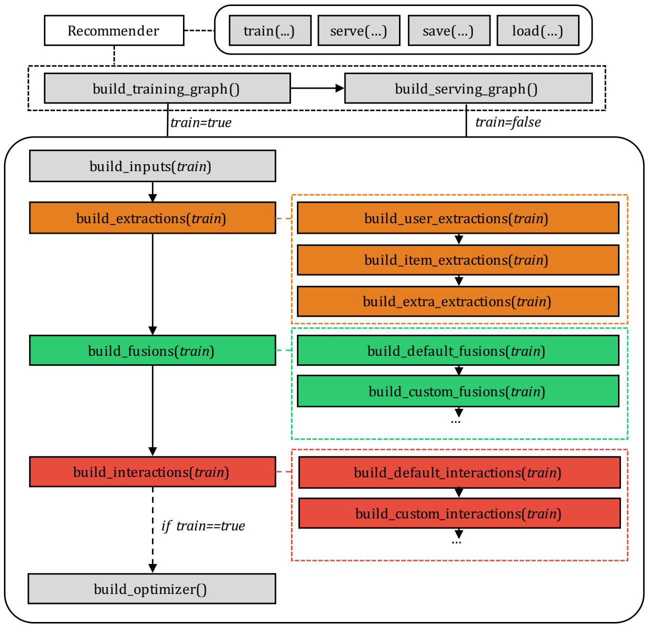
Recommender
train(...)
serve(...)
save(...)
load(...)
build_training_graph()
build_serving_graph()
train=true
train=false
build_inputs(train)
build_extractions(train)
build_user_extractions(train)
build_item_extractions(train)
build_extra_extractions(train)
build_fusions(train)
build_default_fusions(train)
build_custom_fusions(train)
build_interactions(train)
build_default_interactions(train)
build_custom_interactions(train)
if train==true
build_optimizer()

Figure 3: Standard interfaces of the Recommender abstraction. It contains procedures used to construct the computational graph and functions used to drive training, testing and model saving and loading.

into many small steps, so that the system can be easily extended to include new features. As shown in Fig. 3, it consists of two major steps - *build training graph* and *build serving graph*, each of which calls corresponding modules. When building the training graph, a sequence of functions (i.e., *build inputs*, *build extractions*, *build fusions*, *build interactions*, and *build optimizer*) is called with the flag *train* set to *True*, where the extraction, fusion, and interaction modules are built through decomposed child functions. Similarly, when building the serving graph, all of the functions above except *build optimizer* are called with the flag *train* set to *False*. During model training, the function *train* is called for each iteraction; and during testing or evaluation, the function *serve* is used to efficiently score items for a list of users. We show the flexibility and extensibility of the recommender abstraction in Section 5 with concrete examples.

4.2 Modules

Modules represent reusable components in a recommendation algorithm. As discussed in Section 2, a recommender typically contains three components that (1) model the interactions (including ratings, views, likes and thumb-ups, etc.) between users and items in the targeted recommendation context; (2) derive a user's, an item's or a context's representation from a data trace (Fig. 4), such as one-hot encoding, images, text, audio, video, location or demographic information, etc.; and (3) fuse together multiple feature representations from users, items, or environmental contexts. In OpenRec, we name components in these three categories as **interaction**, **extraction**, and **fusion** modules respectively. As shown in Fig. 4, OpenRec modules share the same conceptual architecture and outputs (i.e., a

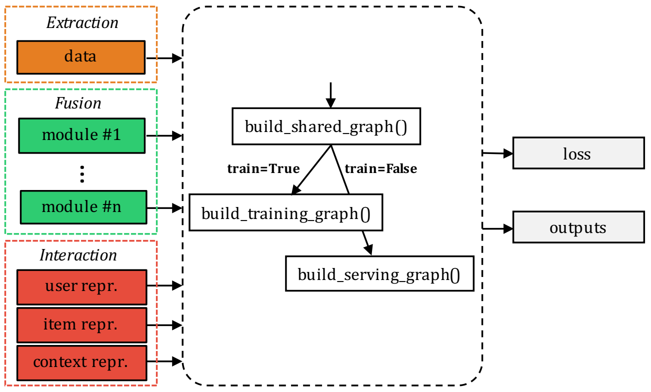

Figure 4: The structure of the Module abstraction. (Left: inputs, Right: outputs)

loss and an output list) but ingest different forms of inputs. Specifically, each module is composed of three core functions: *build shared graph*, *build training graph*, and *build testing graph*. These functions are invoked based on the value of a *train* flag that determines whether we are in training or testing mode.

- **Interaction Module.** An interaction module takes representations from users, items or interaction contexts as inputs and then calculates the loss (during training) and item rankings (during testing). The inputs to the interaction module are typically derived from one-hot encoding or auxiliary information using extraction and fusion modules. The derived loss is used to drive the end-to-end training of the recommender system, and the item rank is used for testing and real-time recommendations. For the interaction module, we do not put any restriction on the number of users and items allowed as inputs so that it is general enough to handle a wide variety of collaborative filtering and content-based algorithms (e.g., Probablistic Matrix Factorization (PMF) is built on pairs of users and items, whereas Baysian Personalized Ranking (BPR) requires triplets of users and items). Our initial prototype of OpenRec includes implementations of many interaction modules using state-of-the-art algorithms, e.g., pairwise logarithm used in BPR [26], pointwise mean square error (MSE) introduced by PMF [27], pairwise euclidean distance adopted in Collaborative Metric Learning (CML) [15], and pointwise cross entropy proposed by Neural Matrix Factorization (NeuMF) [14].
- **Extraction Module.** An extraction module computes representations for a data trace from users, items, or contexts. A simple example is to compute a representation from a one-hot encoding, which performs a basic lookup operation in an embedding matrix. Such a module is leveraged by traditional recommender systems without using auxiliary information, and we refer to it as a Latent Factor module. The development of extraction modules will benefit from advancements in other machine learning fields (e.g., computer vision, natural language processing and speech processing). Models from these fields can be introduced to recommender systems to analyze multi-modal data from users and items. Because OpenRec is highly modular and implemented on Tensorflow, introducing a new content analysis model is rather straightforward and efficient. In our initial prototype, we implemented two general extraction modules, Multi-layer

Perceptron (MLP) and Latent Factor (LF). We expect an open source framework like OpenRec will result in development of more sophisticated models dedicated to analyzing specific data types, such as Convolutional Neural Network (CNN) for images and Recurrent Neural Network (RNN) for sequential data.
- **Fusion Module.** In many recommendation scenarios, users, items, and environmental context may have multiple data sources. For example, in the context-aware recommendation [2] and immersive recommendation [16], a user can be modeled by many personal data traces, e.g., emails, tweets and facebook posts. To bridge the gap between multiple extraction modules and a single interaction module, an fusion module is designed to fuse multiple extraction modules together (Fig. 4). We prototype two intuitive fusion modules, i.e., concatenation and element-wise average.

4.3 Utility functions

In OpenRec, a set of utility functions are included for the ease of model training and evaluation. The model training for recommendation systems usually involves user-item sampling. For example, in BPR, (*user, positive-item, negative-item*) need to be sampled for each training batch. The samplers take the data formatted in Numpy dict as inputs and produce batches of training or validation data for a recommender. We implement popular sampling procedures (e.g. pointwise and pairwise sampling) in the OpenRec framework to drive the training process. In addition, to provide standard model testing, we implement common evaluation metrics (e.g., MSE, Recall@K and AUC) which can be seamlessly integrated with the constructed recommendation model.

4.4 Generalization

Since OpenRec makes few assumption about *users* and *items*, it can be used for a wide range of recommendation techniques and scenarios, e.g., recommendations with different forms of feedback, as well as interactive, conversational, and group recommendation.

- For **different forms of feedback signals**, researchers can customize sampling strategies and interaction modules, for example, using pointwise sampling with the pointwise MSE module for explicit feedback, and pairwise sampling with the pairwise logarithm module for implicit feedback.
- For **interactive and conversational recommender systems**, the optimizers can be designed to update user and item representations in the active-learning settings [34]. For every iteration, a recommender makes recommendations and updates model parameters according to users' and items' representations and their real-time interactions.
- For **group recommendations**, as OpenRec uses Numpy structured arrays as the input data format and does not have restrictions on the number of extraction modules. Users can be grouped based on the additional *group id* inputs to the sampler.

5 EXPERIMENTS AND USE CASES

In this section, we demonstrate the validity, efficiency and extensibility of OpenRec under the following three concrete contexts.

- **Validity.** By comparing the modular implementations with the previous ad-hoc ones, we demonstrate that modularizing recommendation algorithms does not affect the performance and efficiency. Instead, because OpenRec is built on an open-source and industry-standard deep learning tool, it can more efficiently conduct the training. (Section 5.1)
- **Efficiency.** Using OpenRec as a sandbox, developers are able to quickly and efficiently prototype and experiment with different settings of recommender systems and look for the optimal solution (Section 5.2).
- **Extensibility.** By extending and reusing existing modules, OpenRec significantly reduces the overhead of implementing **new recommendation algorithms** (Section 5.3).

For the sake of space, we show the graphical illustration of the modular implementations in the main content and a sample pseudocode snippet in the Appendix A. More examples are available online at http://www.openrec.ai.

5.1 Validity: Reproducing monolithic implementations with modular framework

In order to test whether modularization affects the accuracy and efficiency of the recommendation algorithms, we compare the OpenRec implementations with the implementations released by the original algorithm authors. We use the same model structures and parameter settings from the original papers but replace the training strategies with the standard optimization methods adopted in OpenRec, e.g.. mini-batch stochastic gradient descent. Specifically, we experiment with the following three algorithms in this paper, each of which represents recommender systems with different complexity levels.

5.1.1 Bayesian Personalized Ranking (BPR). As introduced by Rendle et al. [26], Bayesian Personalized Ranking learns latent representations for users and items by using a pairwise ranking loss, as shown in eqn. 1. It is one of the most popular method used under the traditional recommendation context without considering users' and items' auxiliary information.

$$\sum_{(u,i,j)\in D_{\mathcal{S}}} ln\sigma(\hat{x}_{u,i} - \hat{x}_{u,j}) - \lambda_\Theta ||\Theta|| \tag{1}$$

where $\hat{x}_{u,i} = \beta_u + \beta_i + \gamma_u^T \gamma_i$. γ represents a latent representation for an user or an item, and β denotes the corresponding bias term. $D_{\mathcal{S}}$ contains training triplets (u, i, j) where the user u likes the item i but does not indicate her preference for the item j.

To implement the vanilla version of the BPR model using OpenRec, we employ the *Latent Factor (LF)* extraction module to compute latent representations for users and items respectively and a pairwise logarithm interaction module that takes users' and items' representations and computes the loss, which is illustrated in Fig. 5. Note that we do not need to re-implement the existing modules to run the experiment. Building such a recommender system can be achieved by simply putting together the reusable modules with standard interfaces.

We compare the OpenRec modular implementation with the implementation released by He et al. [13] and MyMediaLite library [10] and evaluate them against *tradesy.com* dataset [13], where the products that users *want* and *bought* are treated as positive feedback. As a result, 19,243 users and 165,906 items are included in the experiments. For each user, we randomly sample an item that she likes for validation and another one for testing, which

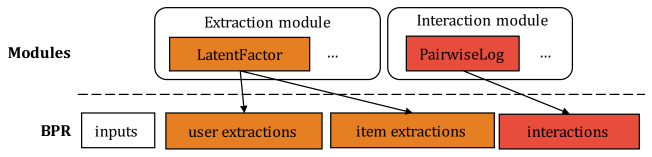

Figure 5: Implementing BPR with OpenRec (45 lines). We use rectangles to represent functions in a Recommender and shade the reusable modules and implementations. An arrow denotes an adoption or an inheritance. (Lines of code does not include blank and import lines)

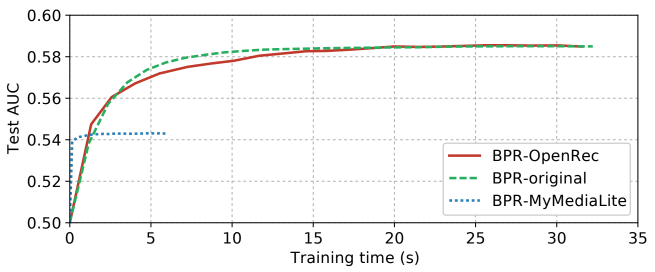

Figure 6: *tradesy.com* dataset [13] testing performance in terms of AUC (BPR-OpenRec, BPR-original, and BPR-MyMediaLite).

is consistent with the strategy used in the original paper [13]. We use the same parameter settings as [13] (λ_Θ is set to 0.1, and the dimensionality of γ is set to 20) and conduct the evaluations on an Amazon EC2 c4.4xlarge instance, which contains 16 CPU cores and 30 GB of memory.

We measure models' performance in terms of Area Under the ROC Curve (AUC), as defined in eqn. 2, against the training time.

$$\text{AUC} = \frac{1}{U}\sum_{u=1}^{U}\frac{1}{|\mathcal{P}(u)|}\frac{\sum_{(u,i)\in\mathcal{P}(u),(u,j)\in\mathcal{N}(u)}\delta(\hat{x}_{u,i} > \hat{x}_{u,j})}{|\mathcal{N}(u)|} \quad (2)$$

where $\mathcal{P}(u)$ contains items the user u likes in the validation/testing dataset, and $\mathcal{N}(u)$ contains items that did not receive any feedback signals from the user u.

The results presented in Fig. 6 show that the modular implementation achieves comparable performance to the best performed BPR implementation, and significantly outperforms the implementation from previous recommendation libraries (MyMediaLite). In other words, modularization does not affect the algorithm accuracy and efficiency for simple models such as BPR.

5.1.2 Visual Bayesian Personalized Ranking (VBPR). The vanilla BPR model does not incorporate any auxiliary information. To investigate recommendation scenarios where such information is leveraged, the second model that we experiment with is Visual Bayesian Personalized Ranking (VBPR), as proposed by [13]. VBPR incorporates visual features into recommendation by learning a transformation function f that projects visual features into the item embedding space. VBPR minimizes the same loss function as BPR but models $\hat{x}_{u,i}$ as follows.

$$\hat{x}_{u,i} = \beta_u + \beta_i + \gamma_u^T\gamma_i + \theta_u^T(\mathbf{E}f_i) \quad (3)$$

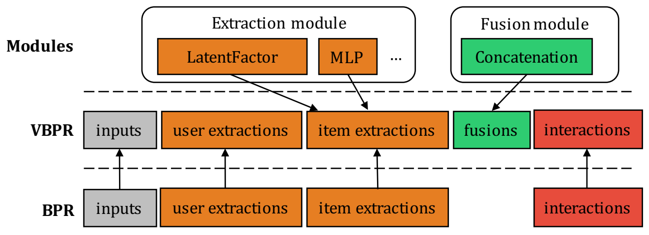

Figure 7: Implementing VBPR with OpenRec (50 lines). We use the same annotations as Fig. 5.

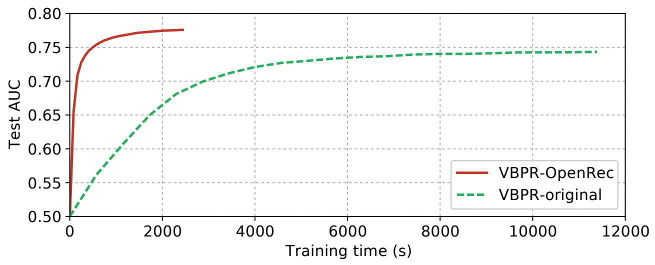

Figure 8: *tradesy.com* dataset [13] testing performance in terms of AUC (VBPR-OpenRec and VBPR-original).

where f_i is the visual feature for item i, and $\mathbf{E}$ is a learnable projection matrix[1].

To build such a model with OpenRec, we can easily extend the BPR recommender and modify the functions *build inputs*, and *build item extractions*. We change the *build item extractions* function from a LF extraction module to a concatenation fusion module that takes as inputs the representations derived from LF and MLP extraction modules, as shown in Fig. 7. At the same time, other functions can be directly reused except adding additional inputs for visual features. We evaluate the VBPR implementation with the same tradesy dataset and computing environment, but set λ_Θ to be 0.1 and the dimensionality of γ and θ to be 10. The items' visual features are extracted using the caffe reference model as released by the He et al. [13].

As shown in Fig. 8, compared to the previous implementation by He et. al. [13], the model implemented by OpenRec is significantly faster (more than 10^3 times) and yields better performance in terms of AUC. The reason for such a phenomenon is that the prior implementation uses a batch size of 1 for the training while OpenRec is able to use much larger mini-batches (batch size is set to 1000) and fully utilize the available hardware resources, e.g. multi-core and GPU, using Tensorflow (We did not use GPU in the experiments for fair comparision). Under the scenario where much auxiliary information is incorporated, the larger batch size brings significant benefits, and OpenRec makes such benefits easily available to the end developers. This example also indicates the need for a benchmarking platform like OpenRec, as directly comparing the performance reported in the literature may be problematic, especially in the cases where some ad-hoc implementation details make significant changes to the recommendation performance.

[1]Compared to the original VBPR, we did not include the visual biases

Table 2: citeulike dataset [31] testing performance in terms of AUC and Recall@K (CDL-OpenRec and CDL-Original).

Implementation	AUC	R@10	R@50	R@100
CDL-OpenRec	0.923	0.107	0.246	0.343
CDL-Original	0.918	0.099	0.248	0.349

5.1.3 Collaborative Deep Learning (CDL). The third algorithm that we explore is Collaborative Deep Learning (CDL), an algorithm built upon the framework of Probabilistic Matrix Factorization (PMF) that uses a de-noising auto-encoder to incorporate text into the recommendations [31]. We refer readers to the original paper [31] for the technical details. Similar to VBPR, CDL can be implemented by extending the PMF recommender, and the extension is analogous to the Fig. 7. We evaluate CDL implementations on the citeulike dataset [31], which contains 5,551 users and 16,980 items and extracts item features using bag-of-words approach. We leverage the same strategy as used in the original paper to split the data into training and testing. The evaluation is conducted under the optimal parameter settings suggested by [31] and on a desktop machine with 8 CPU cores and 16 GB of memory. The performance of each implementation is measured by AUC and Recall@K after convergence. As shown in Table. 2, the results stay consistent with the findings in the previous BPR and VBPR examples - the modular implementation of OpenRec does not degrade the performance and can completely reproduce the results from the original implementations.

5.2 Efficiency: OpenRec as a sandbox for quick prototyping and experimentation

In this section, we show that because of its modular nature, OpenRec can be used as a sandbox for quick designing, prototyping and evaluation in recommendation system research and development. We demonstrate this in the context of building a book recommendation system with rich context and content information, where much information is available from many different channels, including users' purchasing histories, books' content, metadata, user reviews and cover images. Therefore, developers not only need to decide what information to include in the recommender system, but also need to choose appropriate algorithms to analyze data with different modalities. In the rest of this section, we first describe the dataset for experimentation and then show the power of OpenRec in assisting and accelerating such a prototyping process.

5.2.1 Amazon book recommendation dataset. We conduct experiments using an Amazon book recommendation dataset derived from an Amazon review data dump released by [22, 23]. The goal of the system is to recommend books that users are willing to buy. In this experiment, we focus on the utilities of three data sources - users' book purchasing history, users' purchases outside of book category and books' cover images. We include users who have at least 2 purchases in the book category and 5 purchases in non-book categories, which ends up with a dataset containing 99,473 users, 450,166 books and 996,938 purchases. For each user, we derive a **user feature** by taking the bag-of-words representation of the labels for the products purchased in non-book categories. For

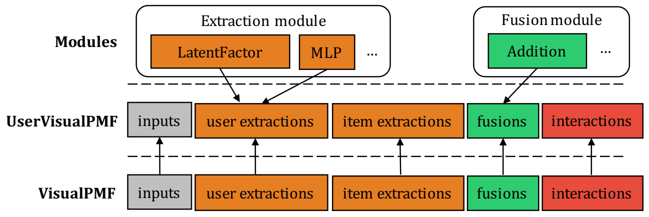

Figure 9: Implementing UserVisualPMF with OpenRec (32 lines). We use the same annotations as Fig. 5.

each book, a **visual feature** is extracted based on the cover image using caffe reference model [19]. We divide the dataset into training/validation/testing by randomly sampling a purchase record for each user for validation and another one for testing.

5.2.2 What information to include? To decide what information to include in the book recommender system, we need to experiment with combinations of the following three data sources: (A) purchasing histories, (B) user features, and (C) visual features - **PMF**(A), **UserPMF**(A+B), **VisualPMF**(A+C), and **UserVisualPMF**(A+B+C). Previously, experimenting on these models required monolithic development for each of them independently, which is a cumbersome and inefficient process. With OpenRec, the UserPMF and VisualPMF are direct extensions of PMF, and the model UserVisualPMF is an extension of UserPMF or VisualPMF. To incorporate users' or items' features, we project them into a low-dimensional embedding space with a multilayer perceptron and treat the outputs as the prior for final representations. In other words, the users' or items' representations are the element-wise addition (fusion) between the projected features and the corresponding latent factors. We implement UserVisualPMF as shown in Fig. 9 (the implementations for VisualPMF and UserPMF are likewise). As the implementation extends most of the functions from VisualPMF and builds additional functions using reusable modules, the overhead of building UserVisualPMF is significantly reduced compared to a monolithic approach. Other fusing strategies such as concatenation are also applicable here, and OpenRec is intuitive in supporting such experiments as well. To compare the performance of these recommender systems, we select the best performed L2 regularization term among $\{0.01, 0.001, 0.0001\}$ using the validation set, and then report the AUC and Recall@K on the testing set. Because of the large number of items, for each user, we randomly sample 1000 items that did not receive any feedback signals to calculate performance metrics.

As shown in Fig. 11a, in terms of AUC, adding visual features or user features significantly improves the recommendation performance, and the best performance is achieved when only visual features are incorporated. However, in terms of Recall@K, the *PMF* model performs relatively well and the *VisualPMF* is able to outperform it when $K \geq 40$. From these results, we can conclude that (1) in general, incorporating auxiliary features is helpful to book recommendations but it does not mean that more features always translate to better performance, and (2) the model selection is contingent on the metric that we want to optimize.

5.2.3 Which algorithm to use? As is studied in the previous experiment, VisualPMF significantly outperforms other systems in terms of AUC. Another interesting question is whether PMF is

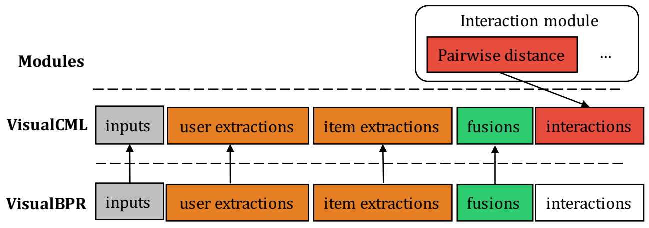

Figure 10: Implementing VisualCML with OpenRec (7 lines). We use the same annotations as Fig. 5. The model and training pseudocode is presented in the Appendix A.

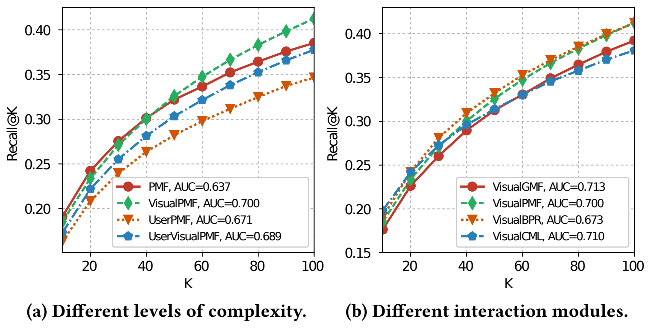

Figure 11: Book recommendation testing performance in terms of AUC and Recall@K.

the best collaborative filtering algorithm under such a recommendation context? We can use OpenRec to quickly investigate this question by leveraging different interaction modules and **reusing the rest of the algorithmic components**. Specifically, we show the performance of **VisualPMF**, **VisualBPR**, **VisualGMF** and **VisualCML** in Fig. 11b. The sample implementation of VisualCML is shown in Fig. 10, which demonstrates that OpenRec provides an elegant and efficient way to quickly experiment with alternative system components.

As shown in the Fig. 11b, varying the interaction module does make a difference in recommendation performance, and the best choice of the interaction module is dependent on the metric that we want to optimize. For example, VisualCML performs the best in terms of Recall@10, while VisualGMF achieves the best ranking performance, i.e. AUC.

The above examples also illustrate that there is no clear-cut solution to design a better recommender system. Design decisions involve trade-offs and require careful experimentation and benchmarking. With the modular design of OpenRec, we are able to support such a development process and allow experimentation of different designs with minimal overhead.

5.3 Extensibility: Developing new algorithms by extending existing modules in OpenRec

In this section, we demonstrate how researchers can use OpenRec to develop new recommendation algorithms by directly extending existing modules. Specifically, we develop a light-weight, iterative

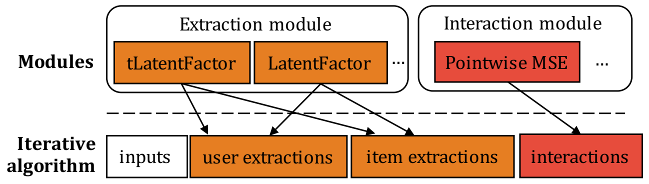

Figure 12: Implementing iterative and temporal model with OpenRec (57 lines). We use the same annotations as Fig. 5.

and temporal recommendation model for movie rating prediction (similar to recent recommendation models [20, 32] that incorporate temporal patterns). We build multi-layer deep neural networks to project user and item vectors from time $t-1$ to t, i.e. $\gamma_u^t = f(\gamma_u^{t-1})$ and $\gamma_i^t = g(\gamma_i^{t-1})$ where f and g are two separate multi-layer perceptrons, and the most recent user and item latent representations are dot-producted to predict the user-item ratings. To develop such a model with previous software frameworks, we would need to build everything from scratch even if there are many existing implementations of matrix factorization and multi-layer perceptrons available. However, using OpenRec, such a temporal model can be built by implementing a new extraction module *tLatentFactor* that executes the transition functions f and g and produces user and item vectors at time t, and directly extending the existing *Pointwise MSE* and *LatentFactor* modules, as Fig. 12 shows. This is possible because of the highly-modular nature of OpenRec. To train the model, we use traditional mean square error as the loss with L2 regularization to drive the optimization. Because OpenRec is built on a Tensorflow backend, benefits such as automatic differentiation are readily available to the developers.

We evaluate our temporal model using the Neflix dataset [3] and compare it to the traditional matrix factorization (MF) implementation from MyMediaLite. We update users' and items' representations daily[2] and validate on each batch before training (Each data point is only used once). The user and item vectors are initialized using MF over the first 3/4 of the dataset (75M ratings). We refer readers to the OpenRec online repo for additional parameter settings. The experimental results demonstrate that our model significantly outperforms the MF baseline by 6% in terms of MSE (0.066 for ours and 0.071 for MF) after only training on 3 days of rating data, which justifies the merits of temporal patterns. Note that our model is not intended to be the state-of-the-art in temporal recommendation but rather as an example of how researchers can easily use OpenRec to explore their ideas.

6 CONCLUSION AND FUTURE WORK

We introduced OpenRec, a modular framework designed to support extensible and adaptable development and research in recommender systems. Through careful experiments and case studies, we demonstrated the value of modularity and reusability. Moving forward, future work will include: standardizing interfaces; building new modules, recommenders, and utility functions (such as NDCG); evaluating models against standard datasets and criteria; and creating modularized models with non-neural network structures (such as random forest).

[2]We only make updates for users and items that have rating during that day.

A VISUALCML OPENREC IMPLEMENTATION

```
from openrec.recommenders import VisualBPR
from openrec.modules.interactions import PairwiseEuDist

class VisualCML(VisualBPR):

    def _build_default_interactions(self, train):
        if train:
            self._interaction_train = PairwiseEuDist(train=True,..)
        else:
            self._interaction_serve = PairwiseEuDist(train=False,..)
```

Listing 1: Pseudocode of an OpenRec implementation for the VisualCML recommender (Section 5.2.3).

```
from openrec import ModelTrainer
from openrec.utils import Dataset
from openrec.recommenders import VisualCML
from openrec.utils.evaluators import AUC
from openrec.utils.samplers import PairwiseSampler

raw_train_data, raw_test_data = load_raw_data()
train_dataset = Dataset(raw_train_data, .., name='Train')
test_dataset = Dataset(raw_test_data, .., name='Test')

model = VisualCML(batch_size=512, ..)
sampler = PairwiseSampler(batch_size=512, dataset=train_dataset)
model_trainer = ModelTrainer(batch_size=512, dataset=train_dataset,
                            model=model, sampler=sampler, ..)
auc_evaluator = AUC()

model_trainer.train(num_itr=1e4, eval_datasets=[test_dataset],
                    evaluators=[auc_evaluator], ..)
```

Listing 2: Pseudocode of training VisualCML with OpenRec.

ACKNOWLEDGMENTS

We would like to sincerely thank the anonymous reviewers for their insightful comments and suggestions. This research is partly funded by Oath through the Connected Experiences Laboratory at Cornel Tech. The work is further supported by the small data lab at Cornell Tech, which receives funding from NSF (#1700832), NIH, RWJF, UnitedHealth Group, Google, and Adobe.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, and others. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Linas Baltrunas, Bernd Ludwig, and Francesco Ricci. 2011. Matrix factorization techniques for context aware recommendation. In *Proceedings of the fifth ACM conference on Recommender systems*. ACM, 301–304.

[3] James Bennett, Stan Lanning, and others. 2007. The netflix prize. In *Proceedings of KDD cup and workshop*, Vol. 2007. New York, NY, USA, 35.

[4] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and others. 2000. Advances in network simulation. *Computer* 33, 5 (2000), 59–67.

[5] Denny Britz, Anna Goldie, Thang Luong, and Quoc Le. 2017. Massive Exploration of Neural Machine Translation Architectures. *ArXiv e-prints* (March 2017). arXiv:cs.CL/1703.03906

[6] Simon Chan, Thomas Stone, Kit Pang Szeto, and Ka Hou Chan. 2013. PredictionIO: a distributed machine learning server for practical software development. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2493–2496.

[7] François Chollet and others. 2015. Keras. https://github.com/fchollet/keras. (2015).

[8] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 191–198.

[9] Michael D Ekstrand, Michael Ludwig, Joseph A Konstan, and John T Riedl. 2011. Rethinking the recommender research ecosystem: reproducibility, openness, and LensKit. In *Proceedings of the fifth ACM conference on Recommender systems*. ACM, 133–140.

[10] Zeno Gantner, Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2011. MyMediaLite: A free recommender system library. In *Proceedings of the fifth ACM conference on Recommender systems*. ACM, 305–308.

[11] Guibing Guo, Jie Zhang, Zhu Sun, and Neil Yorke-Smith. 2015. LibRec: A Java Library for Recommender Systems.. In *UMAP Workshops*.

[12] Ido Guy, Naama Zwerdling, Inbal Ronen, David Carmel, and Erel Uziel. 2010. Social media recommendation based on people and tags. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 194–201.

[13] Ruining He and Julian McAuley. 2015. VBPR: visual bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1510.01784* (2015).

[14] 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*. International World Wide Web Conferences Steering Committee.

[15] 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*. International World Wide Web Conferences Steering Committee.

[16] Cheng-Kang Hsieh, Longqi Yang, Honghao Wei, Mor Naaman, and Deborah Estrin. 2016. Immersive Recommendation: News and Event Recommendations Using Personal Digital Traces. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 51–62.

[17] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. Ieee, 263–272.

[18] Nicolas Hug. 2017. Surprise, a Python library for recommender systems. http://surpriselib.com. (2017).

[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[20] Yehuda Koren. 2010. Collaborative filtering with temporal dynamics. *Commun. ACM* 53, 4 (2010), 89–97.

[21] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing* 7, 1 (2003), 76–80.

[22] Julian McAuley, Rahul Pandey, and Jure Leskovec. 2015. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 785–794.

[23] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. 2015. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 43–52.

[24] Slav Petrov and others. 2016. SyntaxNet. https://github.com/tensorflow/models/tree/master/syntaxnet. (2016).

[25] István Pilászy and Domonkos Tikk. 2009. Recommending new movies: even a few ratings are more valuable than metadata. In *Proceedings of the third ACM conference on Recommender systems*. ACM, 93–100.

[26] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*. AUAI Press, 452–461.

[27] Ruslan Salakhutdinov and Andriy Mnih. 2007. Probabilistic Matrix Factorization.. In *NIPS*, Vol. 1. 2–1.

[28] Yue Shi, Martha Larson, and Alan Hanjalic. 2014. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 3.

[29] Apache Spark. 2017. MLlib. https://spark.apache.org/mllib/. (2017).

[30] Aaron Van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep content-based music recommendation. In *Advances in neural information processing systems*. 2643–2651.

[31] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. 2015. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1235–1244.

[32] Chao-Yuan Wu, Amr Ahmed, Alex Beutel, Alexander J Smola, and How Jing. 2017. Recurrent recommender networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 495–503.

[33] Longqi Yang, Chen Fang, Hailin Jin, Matt Hoffman, and Deborah Estrin. 2017. Personalizing Software and Web Services by Integrating Unstructured Application Usage Traces. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee.

[34] Longqi Yang, Cheng-Kang Hsieh, Hongjian Yang, John P Pollak, Nicola Dell, Serge Belongie, Curtis Cole, and Deborah Estrin. 2017. Yum-me: a personalized nutrient-based meal recommender system. *ACM Transactions on Information Systems (TOIS)* 36, 1 (2017), 7.