BEARS: Towards an Evaluation Framework for Bandit-based Interactive Recommender Systems

Andrea Barraza-Urbina
Insight Centre, Data Science Institute,
National University of Ireland Galway
Galway, Ireland
andrea.barraza@insight-centre.org

Georgia Koutrika
Athena Research Center
Athens, Greece
georgia@imis.athena-innovation.gr

Mathieu d'Aquin, Conor Hayes
Insight Centre, Data Science Institute,
National University of Ireland Galway
Galway, Ireland
firstname.lastname@nuigalway.ie

Abstract

Recommender Systems (RS) deployed in fast-paced dynamic scenarios must quickly learn to adapt in response to user evaluative feedback. In these settings, the RS faces an online learning problem where each decision should optimize two competing goals: gather new information about users and optimally serve users according to acquired knowledge. Related works commonly address this exploration-exploitation trade-off by proposing bandit-based RS. However, evaluating bandit-based RS in an offline interactive environment remains an open challenge. This paper presents BEARS, an evaluation framework that allows users to easily test bandit-based RS solutions. BEARS aims to support reproducible offline evaluations by providing simple building blocks for constructing experiments in a shared platform. Moreover, BEARS can be used to share benchmark problem settings (Environments) and reusable implementations of baseline solution approaches (RS Agents).

1 Introduction

Recommender Systems (RS) help users discover interesting items (e.g., products to buy, movies to watch, restaurants to visit) by means of relevant proactive suggestions. To do so, a RS learns a model of user preferences from user feedback, which is collected over time. In this work, we study RS in an interactive setting, where the system receives interactive feedback from users given sequential recommendations [24]. User feedback can be explicit (e.g., ratings) or implicit (e.g., clicks). In principle, when making recommendations, there are two competing goals that the RS needs to optimize: choosing an action (i.e., a recommendation) that will help learn the user model (*explore*) or choosing an action that will better serve the user (*exploit*). More specifically, item suggestions with the purpose of exploration can be expensive and may reduce user satisfaction in the short term, raising the question of *optimally balancing the two competing goals: maximizing user satisfaction in the long run, and gathering information about the goodness of match between user interests and content.*

Conventional RS typically learn from users' past activity in a supervised learning way and then use the learned model to make recommendations by offering products with the highest predicted utility (i.e., always exploiting). However, in many web-based scenarios (e.g., filtering news articles or display of advertisements), the content universe undergoes frequent changes, with content popularity changing over time as well. Furthermore, a significant number of visitors are likely to be entirely new with no historical consumption record whatsoever; this is known as a cold-start situation. These issues make conventional recommender approaches difficult to apply. In such dynamic and interactive environments, it is essential for the RS method to adapt to the shifting preference patterns of the users and the evolving space of items, a challenge, since most recommendation models are designed to change slowly or offline. Because supervised learning approaches fail to consider the exploration-exploitation trade-off, a possible solution can be found in representing the RS problem as an open-ended Reinforcement Learning (RL) task, where the concepts of exploitation and exploration are key.

In particular, exploration-exploitation methods, a.k.a. *Multi-Armed Bandits* (MAB), have been shown to provide an excellent solution. While applications of MAB to RS are relatively new, already several approaches exist [7, 10, 15, 24]. However, the offline evaluation of methods in a bandit setting is frustratingly difficult [15]. In RS, offline experiments have been commonly based on framing the RS problem as a prediction/classification task. Nevertheless, the dynamic and interactive scenario of the bandit-based RS problem setting introduces several new evaluation challenges compared to the evaluation of a supervised learning model. Mainly, the biggest challenge is that learning is carried out by means of a continuous interaction and active exploration. As a result, the RS learning task is executed in an online fashion, in contrast to supervised learning which uses a fixed set of training data as input [17].

More specifically, standard RS offline evaluation methods make use of datasets, which can either be real-world or artificial, containing user behavioral data used to estimate the quality of a RS. On the one hand, one of the most significant challenges of using real-world datasets for the interactive learning setting is that these ordinarily only represent a small fraction of user preferences, i.e., the dataset holds incomplete ground-truth information [8, 19]. This problem of partial observability or missing values can make it difficult to use real-world datasets to test a learning task based on active exploration. Also, historical information can misrepresent not only the long-term influence of the RS over future user decisions, but also the effects of the dynamic and evolving characteristics of an application domain. On the other hand, to test over an interactive set-up, it is common practice to design simulators or create artificial datasets [9, 16]. Nonetheless, synthetic datasets and simulators can inaccurately model real user behavior and wrongly favor specific algorithms over others [11]. As a consequence, evaluation results may not reflect the RS's real performance and would unavoidably contain modeling biases [16].

Due to the limited existence of standard techniques and benchmarks to conduct offline evaluation for bandit-based RS approaches, past works tend to propose their own evaluation methods and strategies to overcome the challenges discussed above. Alternatively, past

Included in the Offline Evaluation for Recommender Systems Workshop (REVEAL'18), collocated with ACM RecSys 2018.
REVEAL'18, October 6-7, 2018, Vancouver, Canada.
©2018 Copyright held by the owner/author(s).

works have used online evaluation, which is not only highly difficult to replicate but also can be expensive and time-consuming in comparison to offline evaluation methods. All in all, it can be very challenging to reproduce experiments and compare a solution approach to results obtained by prior works in the field of bandit-based RS.

As a solution, the RS field can borrow ideas from the evaluation methods proposed to test Reinforcement Learning (RL) tasks. RL has addressed offline evaluation challenges by sharing simulation environments, as opposed to datasets. As an example, the OpenAI Gym toolkit [3] provides a wide variety of environments that model worlds related to games, robotics, and control tasks, among others.

In this paper, we propose the **BA**ndit-based **R**ecommender **S**ystem **E**valuation Framework, named BEARS, an evaluation framework that allows users to easily test bandit-based RS solutions. BEARS has been designed with a focus on offline evaluation, motivated by the need to offer a common framework that would support the creation of robust and reproducible experiments. Towards this end, BEARS provides simple building blocks and a common platform to share benchmark algorithms and environments. This paper is organized as follows: Section 2 describes the conceptual model that structures the BEARS framework and defines how the RS task is represented as a RL problem in our work. Based on this mapping, we present the BEARS evaluation framework in Section 3. Next, Section 4 describes how to use BEARS for two sample use cases. Finally, we conclude and present future work.

2 BEARS Conceptual Model

The main components of the BEARS evaluation framework depend on representing the Recommender System (RS) task as a sequential decision making problem that balances the exploration and exploitation trade-off. Towards this goal, the BEARS conceptual model focuses on framing the RS task as a Reinforcement Learning (RL) problem (view Section 2.2). Noticeably, in recent years RS works have mostly taken inspiration from solutions that aim to balance the exploration-exploitation trade-off for the Multi-Armed bandits (MAB) problem setting (view Section 2.3), which is a simpler but powerful instantiation of RL. In light of this, though BEARS is inspired by core components used to define a general RL task, we focus the initial development of the framework on the evaluation of bandit-based RS approaches.

In this Section, we first introduce RL and MAB. Next, we present the BEARS conceptual model, which describes how in BEARS the main components defined by the state-of-the-art to represent a RL problem are used to represent an interactive RS task. Lastly, we will briefly discuss a selection of works to illustrate the wide variety of possible bandit-based RS solutions.

2.1 Background

In Reinforcement Learning (RL), a goal-oriented agent (decision-maker) must perform a task while operating in an uncertain environment. To do so, the agent can interact with the environment which offers feedback in terms of positive or negative reward signals, allowing the agent to learn from trial-and-error experiences. This interaction is formalized under the RL framework in Figure 1 [21], where the agent and the environment interact continually in a sequence of discrete time steps t. Every step, the *agent* performs an *action* a_t and the *task environment* presents the agent with its new *state* s_{t+1} and a *reward* r_{t+1} issued due to the executed action. The

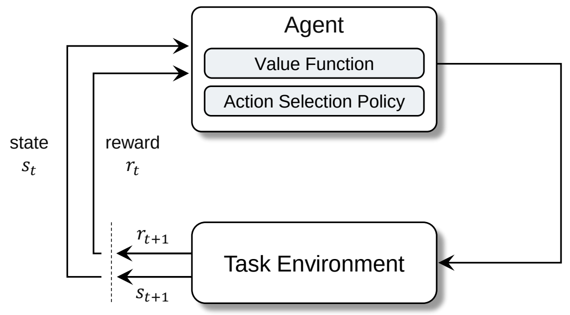

Figure 1: Reinforcement Learning Framework (Based on: [21]).

agent's goal is to learn a *policy*; a strategy indicating the correct action to perform in every state/situation in order to maximize the total reward collected throughout the interaction. In a nutshell, the policy determines how the agent should behave in different situations to achieve its task in an optimal way. It is important to highlight that each decision the agent makes has long-term consequences: each action influences the environment and determines what type of information the agent can observe to update its policy going forward.

The Exploration-Exploitation Trade-off. For illustrative purposes, lets consider an agent seeking to learn the best action to perform in an environment with a single state, so as to maximize the reward it receives after $t \in T$ interactions. In this environment, action rewards are stochastic. Towards learning the optimal policy, the agent could keep an estimate of the expected reward for the different actions it has executed in the past. A simple estimate is the observed average payoff (or empirical average reward) of an action. Lets define the *value function* $Q_t(a)$ as the agent's estimated value for action a at time step t[1]. At a first glance, a good strategy could be to always choose the action with the greatest estimated value $\max_a Q_t(a)$, i.e., the greedy strategy. This is an *exploitative* strategy as the agent is in fact exploiting its current knowledge of the environment. However, an unexperienced agent may be basing its decisions on inaccurate knowledge, and by employing a greedy strategy it would be continuously performing a sub-optimal action. As an alternative, the agent could improve its overall knowledge of action values by exploring non-greedy actions. This *exploratory* strategy could lead to low immediate rewards but it would help the agent refine its knowledge and thus improve its chances of finding the true optimal policy. As can be seen, there is a clear *trade-off between exploration and exploitation* that the agent must balance when making decisions: the agent must explore to search for the optimal action to perform, but as well the agent must use what it has learned and choose the best action as often as possible in order to maximize reward [2]. Indeed, balancing the trade-off between exploitation and exploration is a key and distinctive challenge in RL.

The Multi-Armed Bandit Problem. The Multi-Armed Bandit problem (MAB) is a simplified version of a RL setting, which in its basic form depicts a task environment with a single state. Action selection policies that address the MAB problem (i.e., bandit solutions)

[1]This section offers a a simplified definition of one type of value function. For a more comprehensive definition please refer to [21].

are popular approaches to face the exploration-exploitation challenge in real-world applications (such as RS). The problem depicts a gambler/learning agent faced with the sequential decision making challenge of betting on a K-armed bandit (analogy of a slot machine with K levers, where $K > 1$) for a period of T discrete plays/steps in order to maximize its profit.

In the case of the context-free stochastic MAB, the pay-off or reward offered by each arm $a \in \mathcal{A}$ is a random variable guided by a stationary and independent probability distribution unknown to the gambler. The agent's main goal is to maximize its cumulative reward at the end of the T plays. At each step $t \in T$, the gambler chooses one arm a_t to bet on, and receives as feedback *only* the reward offered by the selected arm. In other words, the agent receives one reward $r_{a_t,t}$ independently sampled from the arm's reward distribution. In this manner, the gambler's knowledge about the environment (i.e., the K-armed bandit) depends on the sequence of actions/bets it chooses during the course of its lifetime.

Different types of *action selection policies* have been developed to balance the exploration-exploitation trade-off. We will highlight three common action selection strategies for stochastic MAB that have been used in RS works (view Section 2.3).

The ϵ*-greedy* policy is a simple approach that selects the greedy action (exploits) with probability $1 - \epsilon$ and selects a random action (explores) with probability ϵ, where $0 \leq \epsilon \leq 1$ [2, 21]. A high value for ϵ might find the optimal action earlier, but might not select it often in the long-run [21]. A low value for ϵ could run the risk of the agent not exploring enough to find the optimal action. A disadvantage of the ϵ-greedy approach is that exploration is random and chooses equally between actions. This strategy could over-explore known inferior actions and overlook promising under-explored actions.

The *Upper Confidence Bound* (UCB) algorithm offers an alternative that explores in accordance to the agent's confidence of estimated values [2]. UCB chooses at time step t the arm a that maximizes the value for $\mathrm{Q_t}(a) + \sqrt{\frac{2 \ln t}{t_a}}$, where t_a is the total number of samples the agent has acquired for arm a at time step t (i.e., the number of times arm a has been played so far) and $\mathrm{Q_t}(a)$ is the agent's estimated value of action a at time step t (i.e., the value function). UCB considers exploitation by favoring arms with high value estimates, and as well considers exploration by favoring arms for which the agent knows less about in comparison to other arms.

Finally, *Thompson Sampling* offers a Bayesian approach towards balancing the trade-off [1, 6, 10, 22]. Given payoffs from past interactions, the posterior reward probability for each arm is estimated. At each iteration, the technique samples each arm's estimated posterior reward probability and uses obtained values as the expected rewards per arm (i.e., to define the value function). Finally, the approach chooses at each step the action that maximizes expected reward.

2.2 The Recommendation Task as a Reinforcement Learning Problem

RL can offer a proper framework (view Figure 1) to represent a RS that learns from evaluative feedback when interacting with an unknown environment. The BEARS evaluation framework, is firmly grounded on the mapping of the RS scenario to the components of the RL framework as follows:

- *Recommender System Task Environment* (*Task Environment*): The *RS Environment* is the domain or application scenario where the agent is deployed. It embodies the problem setting that the agent aims to solve. Thus, a RS Environment represents the dynamics of users and items specific to the application setting. As a rule of thumb, Sutton and Barto [21] define that anything outside of the agent's control is part of its environment.
- *Recommender System Agent* (*Agent*): The RS is a learning agent. At each time step, the *RS Agent* provides a recommendation (i.e, performs an action) given a context/state (e.g. current target user) of the RS Environment. After offering a recommendation (single-item, ranked list), the RS Agent will receive a reward from the RS Environment. In the RS field, rewards are generally a scalar value representative of user explicit/implicit feedback.
- *Value Function:* Traditionally in RS, the core task is to estimate the utility of a recommendation for a user. From a RL perspective, beyond an estimate of the immediate utility/reward of an action, the agent aims to learn from experience an action's long-term value [21]. The action-value function $\mathrm{Q}(s, a)$, defines the value of taking action a in state s. In brief, the value function embodies the agent's goals by quantifying the expected long-term consequences of decisions [21]. A simple representation for $\mathrm{Q}(s, a)$ can be as a look-up table, with an entry for each state-action pair. However, for large-scale environments this representation can quickly become unfeasible. In addition, it is regularly the case that the agent would only have observations over a limited subset of the state-action space. Function approximation methods (Supervised Learning approaches) can be used to define a more compact representation of the estimated value function that can as well generalize to the whole state-action space given a subset of training samples. Using observations retrieved at each time step, the agent updates its estimate of the value function using either: (a) an online learning approach that carries out an incremental update as data becomes available or, (b) a batch learning method executed every $T_u \geq 1$ time steps using data collected in previous time steps.
- *Action Selection Policy:* The action selection policy uses the information provided by the value function to select the next action to execute. The policy must consider that when interacting with an environment the agent has two fundamental but competing goals: to learn how to behave through exploration, and to optimize results by exploiting its acquired knowledge. Section 2.1 describes examples of action selection policies in MAB environments (e.g. UCB and e-greedy).

To clarify, the RS Environment represents the dynamics of users and items across time steps. Note that user and item dynamics can be independent of the RS actions. For example, the item catalog of a RS application can change without any action performed by the RS Agent. Nonetheless, in the case of RS, the environment can be influenced by the decisions of the RS Agent. For example, user behavior (RS Environment) can be guided due to suggestions provided by the RS Agent. In this fashion, the RS Agent can seek to alter its environment through its actions, and as the environment changes, the agent in turn must adapt to meet its goals.

In the following section, we will discuss previous works in the RS field that use bandit strategies. Works were selected to offer a view of the variety of bandit-based approaches.

2.3 Existing bandit-based Recommendation Systems

In general, RS have a rich application domain, and different types of extensions have been made to the basic MAB formulation to tackle specific characteristics and challenges relevant to RS. In this Section, we highlight examples of bandit-based approaches from previous RS works, for both single-item and ranked list recommendation.

Single-Item Recommendation. Hariri et al. [10] propose a RS approach that can adapt to user taste drift by combining a MAB, deployed per user, with a component that is able to detect contextual changes. Wang et al. [23] propose a Bayesian rating model to represent dynamic user musical preferences. Their solution uses Bayes-UCB [12] to learn the parameters of the user preference model. This approach is a content-based RS that deploys one bandit for each user.

Instead of using a single bandit per user as the previous two examples, user preferences can be learned by understanding the rating behavior of user groups.

LinUCB [15] was proposed to address the challenges of the Yahoo!News Today application. The LinUCB approach will be referenced in Section 4 as an example use case to demonstrate the applicability of the BEARS framework. For this reason, in this Section, we take the opportunity to offer slightly more background on this approach. In broad terms, LinUCB is a generalized contextual bandit[2] approach that can consider side information (i.e., contextual features) about users and items in order to maximize total user clicks. In LinUCB with Disjoint Linear Models, the expected payoff for an arm (i.e., the estimated click probability for an item) is a linear model between a context vector $\mathbf{x}_{t,a}$ and an unknown coefficient vector $\hat{\boldsymbol{\theta}}_a$. The coefficient vector is estimated using as training data user-click feedback from previous trials. Uncertainty over the expected payoff is interpreted as a confidence interval over the coefficient vector estimation. Given a measure of uncertainty and an estimated value for an arm, LinUCB's action selection strategy is based on UCB. Specifically, for the Yahoo!News application, contextual features were set to represent how much a user belongs to a given segment of the population that shares item tastes. Thus, it can be interpreted that the coefficient vector learned per arm (per news story), estimates how much a particular user group would like the news article. In this manner, the algorithm is implicitly learning user-group preferences.

Alternatively, to learn about user groups, clustering bandits learn about the user-base clustering structure and at the same time build a preference profile about users according to their assigned cluster. Gentile et al. [7] propose the Cluster of Bandits (CLUB) algorithm, which deploys a bandit strategy per cluster to learn online the number and size of user clusters based on a graph of user social relationships.

Ranked List Recommendation. For ranked list RS, we can extend the classical bandit-based RS in several ways, for example we can define: one bandit per ranking position as in [13, 14], a single bandit where each arm represents a recommendation list or a multiple-play bandit [18]. Kohli et al. [13] propose the independent bandit algorithm (IBA) which optimizes the click-through rate of each ranking position of the recommendation list (i.e., each slot) independently and interprets user feedback based on the probability ranking principle. IBA runs independent bandit instances (e.g., UCB and ϵ-greedy) per slot, which allows for learning to happen in parallel. Louëdec et al. in [18], highlight that approaches for multiple item recommendation based on several independent single-play bandits can only learn about one item of the recommendation list at a time. For this reason, the authors propose to learn about all recommended items at the same time with one multiple-play bandit approach. Towards this goal, the authors propose an adapted version of the *Exp3.M* [4] bandit algorithm.

[2] *Contextual bandits* can use side information about the current state of the environment (i.e., contextual features) to estimate arm/action values [5, 15].

3 BEARS Evaluation Framework

The **BA**ndit-based **R**ecommender **S**ystem **E**valuation Framework (BEARS) aims to support the evaluation of RS solutions built for an interactive problem setting. Namely, the framework is based on the BEARS conceptual model presented as the mapping between RL and RS in Section 2.2. In this work, we present the initial version of BEARS, which is focused on the evaluation of bandit-based RS techniques and not RL-based recommendation approaches in general. BEARS encourages reproducible offline evaluation by providing simple building blocks to execute robust experiments and a common platform. Moreover, BEARS can be used to share benchmark RS Environments and reusable implementations of baseline RS Agents.

The goal of this section is to present the main components of the BEARS framework and to describe how these can be configured to execute experiments. The following section will focus on presenting practical examples that demonstrate how the current version of BEARS can be used. It is important to highlight that the initial version of the BEARS framework will soon be made available as an open-source project developed in Python.

3.1 Architecture Overview: Main Components

The main components of the BEARS framework can be viewed as a class diagram in Figure 2. The framework has been designed to represent a RL problem setting, as described in Section 2, but its applicability in the current version of the framework has only been validated for bandit-based RS approaches. Also, built-in functionalities that support the evaluation of solution approaches have been developed with the requirements of a bandit-based RS task in mind.

As can be seen in Figure 2, the core elements that structure BEARS are the *Experiment*, the *Agent* and the *Environment* components. In BEARS, an *Experiment* is focused on assessing the behavior of an *Agent* interacting with an *Environment* in a sequential decision making process (as described in Section 2). To easily assess the agent's behavior along the interaction based on multiple metrics, BEARS provides the *Evaluator* component (described in more detail in Section 3.3). More specifically, Section 3.2 presents experiments commonly used by prior works defined using BEARS, and how these incorporate the *Evaluator* component to support measuring an agent's performance. Section 3.4 will explain more in depth how an *Agent* is set-up in BEARS and Section 3.5 will focus on how to build *Environments*. Each Section will offer more detail on the inner workings of each of the main component, thereby progressively presenting the different elements of Figure 2.

3.2 Defining an Experiment

In this Section, we define two common experiments used in offline evaluation to test the performance of a bandit-based RS solution approach. Experiments are focused on testing the behavior of a single *Agent* (solution approach) when interacting with a specific *Environment* (when faced with a specific problem setting). Extensions to the defined experiments, for example to test multiple agents simultaneously, are possible but are left for future work.

Overall, the goal of an experiment in BEARS is to measure the agent's performance in terms of specific metrics along the agent-environment interaction. This is because, though it can be argued that in RL the goal of an agent is to maximize its expected cumulative

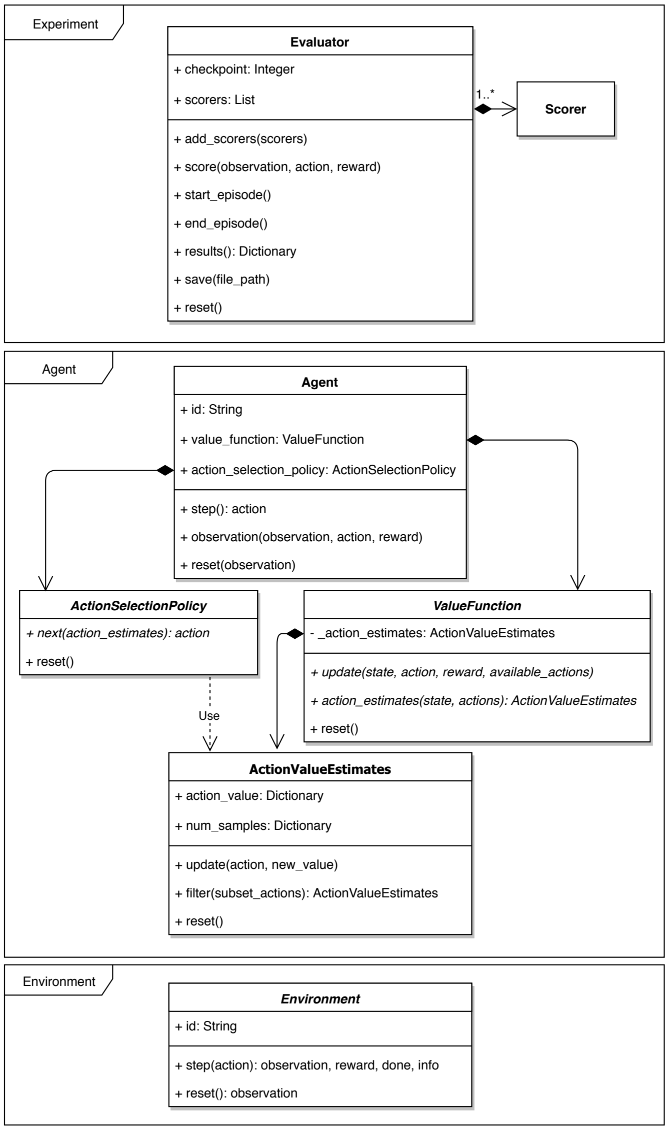
Experiment
Evaluator
+ checkpoint: Integer
+ scorers: List
+ add_scorers(scorers)
+ score(observation, action, reward)
+ start_episode()
+ end_episode()
+ results(): Dictionary
+ save(file_path)
+ reset()
1..*
Scorer
Agent
Agent
+ id: String
+ value_function: ValueFunction
+ action_selection_policy: ActionSelectionPolicy
+ step(): action
+ observation(observation, action, reward)
+ reset(observation)
ActionSelectionPolicy
+ next(action_estimates): action
+ reset()
ValueFunction
- _action_estimates: ActionValueEstimates
+ update(state, action, reward, available_actions)
+ action_estimates(state, actions): ActionValueEstimates
+ reset()
Use
ActionValueEstimates
+ action_value: Dictionary
+ num_samples: Dictionary
+ update(action, new_value)
+ filter(subset_actions): ActionValueEstimates
+ reset()
Environment
Environment
+ id: String
+ step(action): observation, reward, done, info
+ reset(): observation

Figure 2: BEARS Main Components.

reward at the end of all time steps, in the case of RS, beyond the agent's final performance, we are also interested in the continuous assessment of performance throughout all the interactions.

The Base Experiment. A simple experiment is defined in Algorithm 1 by the function *base_experiment* (line 9), which runs a complete process of agent-environment interaction. This experiment follows the intuitive set-up formalized in Section 2, where the agent and environment interact in a sequential loop of T discrete time steps. We will refer to a complete sequence of agent-environment interaction as an *episode*, which is outlined in the function *episode_run* (line 1) in Algorithm 1. Because the *base_experiment* is representative of a single episode, we refer to it as the *Base Experiment.* As input, both the *base_experiment* and the *episode_run*, receive an *Agent* component, an *Environment* component, an *Evaluator* component and the time horizon parameter T.

Algorithm 1 – Python Code
Example Experiments to Assess Agent-Environment Interaction

```
 1: def episode_run(agent, env, evaluator, T):
 2:     for __ in range(T):
 3:         action = agent.step()
 4:         observation, reward, done, info = env.step(action)
 5:         agent.observation(observation, action, reward)
 6:         evaluator.score(observation, action, reward)
 7:         if done then
 8:             break
 9: def base_experiment(agent, env, evaluator, T):
10:     episode_run(agent, env, evaluator, T)
11:     return evaluator.results()
12: def monte_carlo_experiment(agent, env, evaluator, T, E):
13:     for __ in range(E):
14:         evaluator.start_episode()
15:         observation = env.reset()
16:         agent.reset(observation)
17:         episode_run(agent, env, evaluator, T)
18:         evaluator.end_episode()
19:     return evaluator.results()
```

Note that at each time step, the *episode_run* records a snapshot of the current metric values at that point in time by calling the *Evaluator* component (line 6). As a result, the experiment's output is a time series of metric values describing the agent's performance over time. Section 3.3 will further discuss how the *Evaluator* component can collect metric values along time steps.

The Monte-Carlo Experiment. Because an *Agent* and/or an *Environment* can have stochastic properties, we define the *Monte-Carlo Experiment* in function *monte_carlo_experiment* (line 12) in Algorithm 1. As an example, an *Agent* could have stochastic behavior when it uses a random policy to explore. In a similar fashion, an *Environment* is stochastic when the execution of a specific action in a specific state generates a different reward at different trials.

The Monte-Carlo Experiment relies on repeated sampling of episode results to interpret how an *Agent* would generally perform when interacting with a particular type of *Environment*. In other words, we want to consider the average performance of the *Agent* over several sample episodes. The Monte-Carlo Experiment executes E episodes, and provides summary statistics about the overall performance of the agent across the different episode runs by using the *Evaluator* component. To achieve this, the *Evaluator* component is informed when an episode execution will start (line 14) and when an episode execution has ended (line 18). Section 3.3 will further explain how the *Evaluator* component estimates the general performance of an agent given several sample episode results.

This section has highlighted two example offline experiments to evaluate bandit-based RS approaches. Though we cannot provide an exhaustive high level categorization that can characterize all types of possible experiments, the framework is flexible enough that it can be further extended to include future state of the art evaluation methods. Particularly, as an immediate next step the BEARS framework aims to incorporate the offline policy evaluator based on rejection sampling proposed in [15, 16], which can be achieved as a simple modification of the Base Experiment, and BRED [20], an evaluation strategy based on bootstrapping techniques.

3.3 The Evaluator Component

The *Evaluator* component is an essential piece of the BEARS framework that supports collecting metric values throughout an experiment as described in Section 3.2. The core functions and attributes of the *Evaluator* component can be viewed in Figure 2. Mainly, the *Evaluator* component depends on an array of *Scorer* components to measure performance using different metrics. Figure 3 provides a class diagram of the elements that make up a *Scorer* component.

In the Base Experiment defined in Section 3.2, the *Evaluator* component receives at each iteration of an episode the information it requires to make an assessment of the agent's performance at that particular time step (line 6 in Algorithm 1). It broadcasts this information to its array of *Scorers* every *checkpoint* time steps, by calling the function *add_checkpoint* of the *Scorer* (view Figure 3). When a checkpoint is added, each *Scorer* quantifies the agent's performance by using the received information and its assigned *Metric* component. Then, each *Scorer* aggregates the obtained metric value with the score from the previous checkpoint using the *TimeAggregator* component. The *TimeAggregator* component is used to incrementally update metric values between checkpoints towards getting the new checkpoint score. Currently, there are three implemented strategies that can be used as time aggregators:

- *DefaultTimeAggregator*: Does not aggregate the new metric value with the previous score. Thus, the new score is the time step's metric value.
- *AvgTimeAggregator*: Keeps an incremental mean score of metric values received in previous time steps and the new metric value. The new score is the mean score of metric values captured up to the current time step.
- *CumulativeTimeAggregator*: Adds the new metric value to the previous score to get the new score for the current time step. Thus, the new score is the cumulative sum of metric values captured up to the current time step.

The new score is annexed to the *episode_sample* list, which at the end of an episode will hold time aggregated metric values for each checkpoint iterations. In this fashion, by using the *Evaluator* component an experiment can automatically capture a time series of scores, per metric of interest, that can describe the agent's performance in time.

In the case of the Monte-Carlo Experiment defined in Section 3.2, the goal is to capture the agent's performance across different episode runs. To achieve this, the *Evaluator* is informed about the start and end of an episode through the functions *start_episode* and *end_episode* correspondingly (view Figure 3). When the end of an episode is reached, the *Evaluator* aggregates the time series of scores it has collected about the sample episode, i.e., the *episode_sample* list, to its summary statistics using the *EpisodeAggregator* component. Then, the *episode_sample* list is emptied so as to initiate the collection of scores for a new episode run. The *EpisodeAggregator* component uses the sample episode time series to incrementally update online statistics (e.g., mean value, variance, among others) about the scores the *Agent* has achieved at the different checkpoint time steps across episodes.

Overall, setting up a *Scorer* implies assigning a *Metric*, *TimeAggregator* and *EpisodeAggregator* component. The BEARS framework already incorporates useful built-in instantiations of the *Metric* component (e.g., *MeanSquarredError*) and the *TimeAggregator* component (view Figure 3). Nonetheless, we aim to extend BEARS into a toolkit that contains a more comprehensive selection of possible metric functions. The provided *EpisodeAggregator* component can incrementally update statistics as new episode samples arrive, and does not require keeping a collection of samples to generate statistics in batch at the end of an experiment. Also, the *EpisodeAggregator* can be customized to keep track of only the statistics that are relevant for a specific evaluation. This makes the data collection process more efficient and allows to create experiments that run more episodes with larger time horizons. All in all, the modularity and flexibility of creating *Scorer* components allows for the creation of robust experiments that can measure different aspects of the agent-environment interaction.

The results obtained by an *Evaluator* instantiation are returned in a dictionary object that can be easily consulted. We will refer to this dictionary as *results*. Each *Scorer* is assigned a unique identifier key, which is derived from mixing the identifier of its assigned *Metric* and *TimeAggregator* components. When consulting *results*[*scorer_id*] the user finds a sub-dictionary whose keys are the identifiers of the collected summary statistics by the *EpisodeAggregator* component (e.g., mean, std) of the specific *Scorer*. The BEARS open-source project documentation will provide a list of *Scorer* identifiers for the possible scorers, as well as identifiers for the possible summary statistics of the *EpisodeAggregator* component. This does not mean that the framework's use is limited to existing scorers as the developer can easily create their own instantiations of any component as long as the provided interfaces are respected.

3.4 Building an Agent

Following the conceptual model described in Section 2, an *Agent* in BEARS is composed of two main components: a *Value Function* and an *Action Selection Policy*. In a nutshell, the experiences or observations collected by the agent when interacting with an environment help the agent keep an updated estimate of the potential payoff or expected reward of executing an action in a particular environment state. This knowledge about actions is modeled by the agent's value function. To select the next action to perform based on the estimated action values, the agent uses an action selection policy to make decisions in a manner that balances both exploration and exploitation goals.

In BEARS, when an *Agent* component performs an action the *Environment*'s feedback is received through the *observation* function (view Figure 2, the *Agent* class), which has the following input parameters:

- *observation*: relevant information describing the *Environment* and its new state after the action was performed. A RS environment would typically return in the *observation* object the new target user and the available items that can be recommended at the new time step.
- *action*: the action executed in the *Environment*.
- *reward*: the reward issued due to the performed *action*.

This information is used to update the agent's value function, and thus generate more reliable estimates of the expected payoff of actions. The agent's *ValueFunction* component is in charge of keeping track of the agent's current knowledge about action value estimates by updating its model every time a new observation is received through the *update* function (view Figure 2, the *ValueFunction* class).

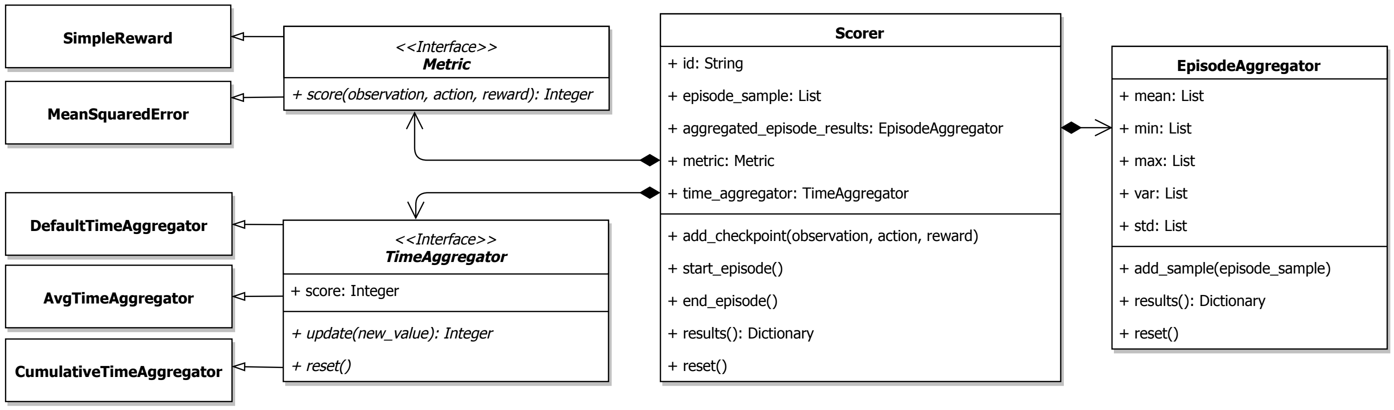

Figure 3: BEARS Scorer.

When the *Agent* is requested to make a decision towards an action to perform (line 3, Algorithm 1), the *Agent* first consults the *ValueFunction* to obtain an *ActionValueEstimates* object. This object is returned by the function *action_estimates* of the *ValueFunction* (view Figure 2, *ValueFunction* class) which receives as input the available actions the *Agent* can perform and the current state of the *Environment*. The returned *ActionValueEstimates* object holds all the necessary information, limited to the actions the *Agent* can perform at time step t, to allow the *ActionSelectionPolicy* component to select the specific next action to perform in the interaction. In short, the *Agent*'s decision is obtained by calling the function *next* of the *ActionSelectionPolicy* which receives as input the *ActionValueEstimates* object generated by the *ValueFunction*. As explained in Section 2, an action selection policy uses the action value estimates to make a decision over which action to perform in a way that balances exploration and exploitation goals.

As can be observed, the *ActionValueEstimates* object acts as a bridge between the *ValueFunction* component and the *ActionSelectionPolicy* component, which can each be developed independently. This makes it very easy to set-up variants of *Agent* solutions using BEARS.

3.5 Building an Environment

The *Environment* component is the domain model, and it represents the dynamics of users and items specific to the application setting (e.g., benchmark problems, real-world applications). In a way, it is a simulation of how the real-world would respond given a sequence of actions/recommendations by the agent. Nonetheless, BEARS is not focused on providing strategies to create a full fledged RS simulation. The goal of BEARS is to provide a common interface that can be used to create benchmark *Environments* that can interact with BEARS *Agents*. The *Environment* definition in BEARS takes direct inspiration to the definition of environments offered by the OpenAI Gym toolkit [3], which has been well validated and shown to work for other RL applications different from RS. In the future, we hope to make the two frameworks compatible, and in this way, invite the RL community to also contribute towards solutions that address the RS problem.

In Figure 2, the definition of attributes and functions of the *Environment* component can be found. The component provides a simple interface where the main function is *step*. This function receives an action to perform from the *Agent* and returns to the *Agent* the following feedback: *observation, reward, done* and *info*. The definitions for the parameters *observation* and *reward* are the same as those received by the *Agent* in the *observation* function (view Section 3.4). The parameter *done* indicates that the *Environment* has reached a goal/final state. Finally, the parameter *info* offers useful information about the *Environment* that can be used for debugging purposes.

An *Environment* component can also be thought of as the definition of general rules and dynamics that guide an *Environment* class or family of possible environments. For example, we could design a simple general domain *Environment* class that generates rewards for arms/actions from independent normal distributions. A particular instantiation of the *Environment* class would establish specific parameter values to define the arm distributions, where these values can be assigned randomly. In this case, the goal of the Monte-Carlo Experiment (view Section 3.2) would be to estimate how the *Agent* performs when interacting with different instantiations of environments characterized by the properties of the *Environment* class. Though the experiment cannot test all possible environments that can be generated from the *Environment* class, it aims to test over a sufficient number of sample environments across the different episodes to estimate overall performance. This notion is illustrated in line 15 of Algorithm 1, where the function *reset* could return a new sample *Environment* from the *Environment* class.

Finally, it is important to highlight that though OpenAI Gym [3] is a valuable framework to build and share environment definitions, the existing toolkit focuses on different applications from RS. In addition, even if the BEARS *Environments* could be fully represented using OpenAI, the BEARS framework remains relevant as OpenAI does not provide tools towards building a solution approach (BEARS *Agent* component) or to keep track of metrics (BEARS *Evaluator* component).

4 Two Sample Use Cases using BEARS

This section aims to show how to build two *Agents* using BEARS. The *first use case* aims to implement a simple bandit approach that uses the observed average payoff (or empirical average reward) of an action as its action value estimate, and an ϵ-greedy action selection policy to select the next action to perform (view Section 2). The simple bandit *ValueFunction* and *ActionSelectionPolicy* can be viewed in Algorithm 2.

Algorithm 2 – Pseudocode
Simple Bandit *ValueFunction* and *ActionSelectionPolicy* using BEARS

```
class AverageValueFunction (ValueFunction):
    $\_action\_estimates = ActionValueEstimates()$
    def reset():
        $\_action\_estimates$.reset()
    def update ( _ , action, reward, _ ):
        if $action \in \_action\_estimates$ then
            $old\_value = \_action\_estimates$.action_value[$action$]
            $n = \_action\_estimates$.num_samples[$action$]
        else
            $old\_value = 0$
            $n = 0$
        $new\_value = old\_value + \frac{1}{n+1} \cdot (reward - old\_value)$
        $\_action\_estimates$.update($action$, $new\_value$)
    def action_estimates ( _ , actions ):
        return $\_action\_estimates$.filter($actions$)
class EpsilonGreedy (ActionSelectionPolicy):
    $\epsilon \in [0, 1]$
    def next ($action\_estimates$):
        $coin$ = Generate a random number between 0 and 1
        if $coin < \epsilon$ then
            $a$ = Selects random $a \in action\_estimates$
        else
            $a = \arg\max_{a \in action\_estimates} action\_estimates$
        return $a$
```

The *second use case* aims to implement the LinUCB approach with Disjoint Linear Models proposed in [15] (view Section 2.3). In this case, to create the agent's *ValueFunction*, we took Algorithm 1 defined in [15] and adapted it to the structure provided by the BEARS framework. The result can be found in Algorithm 3. According to [15], the value for $p_{t,a}$ (line 18, Algorithm 3) gives a reasonably tight upper confidence bound (UCB) for the expected payoff of action a. Because the action value estimates are actually the UCB estimates for actions, then the *ActionSelectionPolicy* is a greedy strategy.

Algorithm 3 – Pseudocode
LinUCB [15] *ValueFunction* and *ActionSelectionPolicy* using BEARS

```
class LinUCBValueFunction (ValueFunction):
    $\alpha \in \mathbb{R}_+$
    Model Parameters $\mathbf{A}, \mathbf{b}$
    def reset():
        Reset Model Parameters $\mathbf{A}, \mathbf{b}$
    def update (state $\mathbf{x}_{t,a}$, action $a$, reward $r$, _ ):
        if $a$ is new then
            $\mathbf{A}_a \leftarrow \mathbf{I}_d$
            $\mathbf{b}_a \leftarrow \mathbf{0}_{d\times 1}$
        $\mathbf{A}_a \leftarrow \mathbf{A}_a + \mathbf{x}_{t,a}\mathbf{x}_{t,a}^\top$
        $\mathbf{b}_a \leftarrow \mathbf{b}_a + r\mathbf{x}_{t,a}$
    def action_estimates (state $\mathbf{x}_{t,a}$, available actions $\mathcal{A}_t$):
        for all $a \in \mathcal{A}_t$ do
            if $a$ is new then
                $\mathbf{A}_a \leftarrow \mathbf{I}_d$
                $\mathbf{b}_a \leftarrow \mathbf{0}_{d\times 1}$
            $\hat{\theta}_a \leftarrow \mathbf{A}_a^{-1}\mathbf{b}_a$
            $p_{t,a} \leftarrow \underbrace{\hat{\theta}_a^\top \mathbf{x}_{t,a}}_{\text{Expected Value}} + \alpha \underbrace{\sqrt{\mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a}}}_{\text{Model Uncertainty}}$
        $\_action\_estimates$ = ActionValueEstimates ($\mathcal{A}_t$)
        for all $a \in \mathcal{A}_t$ do
            $\_action\_estimates$.update($a$, $p_{t,a}$)
        return $\_action\_estimates$
class Greedy (ActionSelectionPolicy):
    def next ($action\_estimates$):
        return $\arg\max_{a \in action\_estimates} action\_estimates$
```

5 Conclusion and Future Work

Recommender Systems (RS) depend on user evaluative feedback, collected over time, to continuously learn a model of user tastes and preferences. In dynamic and interactive problem settings, RS face an exploitation-exploration trade-off when making decisions. Historically, this trade-off has been an important topic of research in the field of Reinforcement Learning (RL), and novel solution approaches can be found when framing the RS task as a RL problem. More specifically, to address the exploration-exploitation trade-off, recent RS works have been proposed based on Multi-Armed Bandit (MAB) techniques. Nonetheless, standard RS evaluation methods inspired by the Supervised Learning field have proven to be inappropriate to assess a sequential decision-making scenario, such as the one faced by the bandit-based RS. As a result, there is a lack of common tools and benchmarks to support the offline evaluation of these new approaches, and reproducing results from prior works can be an important challenge.

This paper presents BEARS (short for **BA**ndit-based **R**ecommender **S**ystem **E**valuation Framework), an evaluation framework written in Python that allows users to easily build experiments aimed at testing bandit-based RS solutions. Currently, the framework provides simple building blocks that allow to easily configure variants for Agents (solution approaches) and Environments (problem settings). More importantly, BEARS allows users to quickly set-up experiments focused on quantifying the performance of an Agent, when interacting with an Environment, in terms of different metrics in a time-aware fashion.

Our vision for the BEARS evaluation framework is to be used as a common platform to support the implementation and sharing of not only solution approaches, but also problem settings. This means, in addition to the provided framework, we aim to also offer a set of built-in benchmark Agents and Environments that can be used out of the box to facilitate the evaluation of new interactive RS approaches. Moreover, in future versions, we aim to provide a comprehensive suite of implementations for common experiments, metrics and useful visualizations to enable a faster, more robust and replicable evaluation process. Towards achieving our vision, we believe that the support of the RS community will be instrumental, and thus we plan to soon make the project open-source. Finally, the MAB setting is a simple instantiation of the full-fledged RL problem. Our long-term focus is to validate the use of the BEARS framework to represent RS approaches based on RL concepts beyond the bandit approach.

Acknowledgments

This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289, co-funded by the European Regional Development Fund.

References

[1] Shipra Agrawal and Navin Goyal. 2013. Thompson Sampling for Contextual Bandits with Linear Payoffs.. In *ICML (3)*. 127–135.

[2] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2-3 (2002), 235–256.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI gym. *arXiv preprint arXiv:1606.01540* (2016).

[4] Sébastien Bubeck, Nicolo Cesa-Bianchi, and others. 2012. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning* 5, 1 (2012), 1–122.

[5] Giuseppe Burtini, Jason Loeppky, and Ramon Lawrence. 2015. A survey of online experiment design with the stochastic multi-armed bandit. *arXiv preprint arXiv:1510.00757* (2015).

[6] Olivier Chapelle and Lihong Li. 2011. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*. 2249–2257.

[7] Claudio Gentile, Shuai Li, and Giovanni Zappella. 2014. Online clustering of bandits. In *International Conference on Machine Learning*. 757–765.

[8] Frédéric Guillou, Romaric Gaudel, and Philippe Preux. 2016. Scalable Explore-Exploit Collaborative filtering. In *PACIS*. 309.

[9] Asela Gunawardana and Guy Shani. 2015. Evaluating recommender systems. In *Recommender Systems Handbook*. Springer, 265–308.

[10] Negar Hariri, Bamshad Mobasher, and Robin Burke. 2015. Adapting to User Preference Changes in Interactive Recommendation.. In *IJCAI*. 4268–4274.

[11] Jonathan L Herlocker, Joseph A Konstan, Loren G Terveen, and John T Riedl. 2004. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)* 22, 1 (2004), 5–53.

[12] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. 2012. On Bayesian upper confidence bounds for bandit problems. In *Artificial Intelligence and Statistics*. 592–600.

[13] Pushmeet Kohli, Mahyar Salek, and Greg Stoddard. 2013. A Fast Bandit Algorithm for Recommendation to Users With Heterogenous Tastes.. In *AAAI*.

[14] Anisio Lacerda. 2017. Multi-Objective Ranked Bandits for Recommender Systems. *Neurocomputing* 246 (2017), 12–24.

[15] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. 2010. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*. ACM, 661–670.

[16] Lihong Li, Wei Chu, John Langford, and Xuanhui Wang. 2011. Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM, 297–306.

[17] Michael L Littman. 2015. Reinforcement learning improves behaviour from evaluative feedback. *Nature* 521, 7553 (2015), 445–451.

[18] Jonathan Louëdec, Max Chevalier, Josiane Mothe, Aurélien Garivier, and Sébastien Gerchinovitz. 2015. A Multiple-Play Bandit Algorithm Applied to Recommender Systems.. In *FLAIRS Conference*. 67–72.

[19] Jérémie Mary, Romaric Gaudel, and Preux Philippe. 2014. Bandits Warm-up Cold Recommender Systems. *arXiv preprint arXiv:1407.2806* (2014).

[20] Jérémie Mary, Philippe Preux, and Olivier Nicol. 2014. Improving offline evaluation of contextual bandit algorithms via bootstrapping techniques. In *International Conference on Machine Learning*. 172–180.

[21] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

[22] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3/4 (1933), 285–294.

[23] Xinxi Wang, Yi Wang, David Hsu, and Ye Wang. 2014. Exploration in interactive personalized music recommendation: a reinforcement learning approach. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 11, 1 (2014), 7.

[24] Xiaoxue Zhao, Weinan Zhang, and Jun Wang. 2013. Interactive collaborative filtering. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*. ACM, 1411–1420.