Spring 5-2018

Assessing the Quality and Stability of Recommender Systems

David Shriver
University of Nebraska-Lincoln, dlshriver@huskers.unl.edu

ASSESSING THE QUALITY AND STABILITY OF RECOMMENDER SYSTEMS

by

David Shriver

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Sebastian Elbaum

Lincoln, Nebraska

May, 2018

ASSESSING THE QUALITY AND STABILITY OF RECOMMENDER SYSTEMS

David Shriver, M.S.

University of Nebraska, 2018

Adviser: Sebastian Elbaum

Recommender systems help users to find products they may like when lacking personal experience or facing an overwhelmingly large set of items. However, assessing the quality and stability of recommender systems can present challenges for developers. First, traditional accuracy metrics, such as precision and recall, for validating the quality of recommendations, offer only a coarse, one-dimensional view of the system performance. Second, assessing the stability of a recommender systems requires generating new data and retraining a system, which is expensive.

In this work, we present two new approaches for assessing the quality and stability of recommender systems to address these challenges. We first present a general and extensible approach for assessing the quality of the behavior of a recommender system using logical property templates. The approach is general in that it defines recommendation systems in terms of sets of rankings, ratings, users, and items on which property templates are defined. It is extensible in that these property templates define a space of properties that can be instantiated and parameterized to characterize a recommendation system. We study the application of the approach to several recommendation systems. Our findings demonstrate the potential of these properties, illustrating the insights they can provide about the different algorithms and evolving datasets.

We also present an approach for influence-guided fuzz testing of recommender system stability. We infer influence models for aspects of a dataset, such as users or items, from the recommendations produced by a recommender system and its training data. We define dataset fuzzing heuristics that use these influence models for generating modifications to an original dataset and we present a test oracle based on a threshold of acceptable instability. We implement our approach and evaluate it on several recommender algorithms using the MovieLens dataset and we find that influence-guided fuzzing can effectively find small sets of modifications that cause significantly more instability than random approaches.

ACKNOWLEDGMENTS

I would like to acknowledge and thank all of the wonderful people without whom this work would not have been possible. None of this would have been possible without the support of my advisor, Sebastian Elbaum. I cannot thank him enough for the guidance and encouragement he has provided over the past several years. I would also like to thank Matt Dwyer and David Rosenblum for their discussion and collaboration on this research. Thank you as well to Gregg Rothermel for being a part of my committee. Thank you to all of my friends and colleagues in ESQuaReD for all of the support, motivation, and discussion over the past two years. Finally, a special thank you to my family for all of their love and support.

GRANT INFORMATION

This work has been supported in part by National Science Foundation Award #1526652.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recommender systems filter information to help users make decisions when lacking personal experience or knowledge [47] or when the set of choices is overwhelmingly large [28]. In general, they take in data about users' past preferences for items and aggregate it to predict a user's preference for unknown items, often presenting a list of the items most likely to be preferred. We see them everywhere, from e-commerce sites such as Amazon, to news article recommendation at the New York Times, and movie recommendations by Netflix. Assessing these systems, for which recommendations can affect revenue streams worth billions of dollars, is challenging. Developers must assess multiple aspects, including two that are key: the quality of recommendations, and the stability of a recommender system as its rating set changes.

The quality of a recommender system depends on many properties of the recommendations it produces, which are typically assessed using performance metrics that capture a narrow view of system behavior. Most commonly, developers assess the accuracy of recommendations with metrics that measure the ability of a recommender system to faithfully represent a set of known preferences. Accuracy is measured by reserving a portion of a ratings dataset as a set of known preferences, and using the rest of the ratings to train a recommender system and make recommendations. The known preferences can then be used by some metric, such as precision or recall, to compute a score representing the accuracy of the recommender algorithm on that dataset.

However, accuracy metrics only provide a coarse, one-dimensional view of the performance of a recommender system. In particular, precision and recall are limited in several ways. First, they provide little information or intuition about the behavior of items with unknown preferences. For instance, if the set of known liked movies for a user is *Toy Story* and *Cast Away*, then the two Top-3 ranked lists of *Toy Story*, *The Incredibles*, *A Bug's Life*, and *Cast Away*, *Forest Gump*, *Saving Private Ryan*, would receive the same precision and recall. However, these two ranked lists represent very different behavior, which is not characterized by common accuracy metrics, but may be useful for a developer in deciding which recommender to choose. Second, when the set of known liked items is small relative to the number of available items, top-k precision and recall values are typically low, due to the small likelihood of recommended items being included in the set of known preferences. This can make it difficult to determine whether low values of precision and recall should be satisfying or deeply troubling. Finally, because precision and recall do not convey any intuition about the behavior of items which are not in the set of known preferences, conclusions about the relative superiority of one system over another can be misleading.

Due to the limited intuitions provided by accuracy metrics, researchers have explored other properties of recommender systems that can be useful to developers beyond recommendation accuracy, as well as metrics to capture them. Several such properties, often with corresponding metrics, have been identified in the recommender systems literature, such as coverage [22, 27], diversity [61, 62], novelty [43, 57], and adaptivity [53]. For instance, recommendations that are diverse—in which items are different from each other—may provide more value to a user. If a user is known to like Sci-Fi movies, recommending all of the Star Wars movies may be accurate, but it is not diverse and may be unsatisfying to a user. Prior work studying properties of recommender systems does not formally define a broad space of properties, typically defining properties narrowly with a formal metric, or describing them informally.

The second challenge we focus on is assessing stability. The stability of a recommender system measures the consistency of the recommendations after changes are made to the dataset [5]. For example, consider a recommender system that suggests the movie *Interstellar* as the top ranked item to 1% of users. If some user adds a new rating for *Interstellar* and as a result it is no longer recommended to any user, then the recommender system could be considered unstable and negatively impact user confidence. The stability of a recommender system can be an expensive property to measure without extensive knowledge of the recommender algorithm, as it requires generating dataset modifications, training a recommender with the modified data, and measuring the distance between the new and the original recommendations. Furthermore, the space of potential dataset modifications is enormous and is infeasible to exhaustively explore.

In this work we address the problems of assessing the quality and stability of recommender systems. Because assessing the stability of a recommender system requires the generation of new datasets, we break our approach into two pieces, as shown in Figure 1.1. To assess recommender system quality we present an approach for characterizing recommender system behavior through the instantiation of logical property templates. We define a general model of recommender systems which we then use to formally define and systematically explore a space of properties of recommender systems in terms of the relations between users, items, ratings and rankings. To assess the stability of a recommender system we present influence-guided fuzzing. We infer influence models from a dataset and recommendations, and use these influence models to fuzz modified datasets. We then train a new recommender system using the modified dataset and check stability with a differentiating oracle.

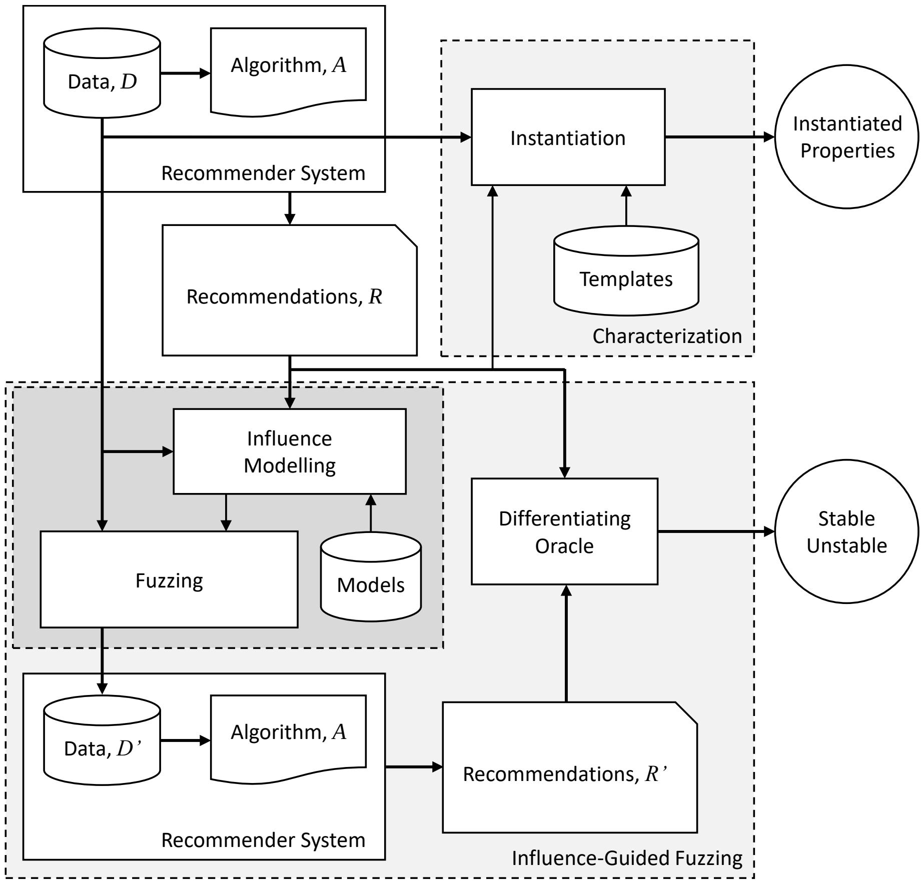

Figure 1.1: Our approach to assessing the quality and stability of a recommender system. We present two approaches (in gray). The first assesses the quality of a recommender system by instantiating logical property templates. The second assesses the stability of a recommender system by using influence-guided fuzzing to generate modified datasets, which are used to train a new recommender system. Stability is assessed by comparing the original recommendations to the new recommendations with a differentiating oracle.

1.1 Contributions

The overall contributions of our work are:

- We present an approach to assessing the quality of recommender systems that provides a more nuanced yet rigorous view of their comparative strengths and weaknesses while also revealing anomalous behavior that is at odds with their coarse-grained summary statistics such as precision and recall.

- We evaluate our quality assessment approach using several recommender algorithms and datasets and find that our approach provides insights into the behavior of recommender systems that are complementary to precision and recall. For instance, we show that a known hybrid recommendation algorithm, trained on the MovieLens dataset, has superior precision and recall to a known model-based algorithm trained on the same dataset. However, the model-based system is able to recommend 3.5 times more unique items and is able to provide a unique set of recommendations for almost every user.

- We present an influence-guided fuzzing approach for the validation of recommender system stability. We define several models that infer influence from a dataset and recommendations, as well as several dataset fuzzing heuristics. We implement our approach in a tool for generating modification sets that are more likely to cause instability.

- We evaluate influence-guided fuzzing on several recommender algorithms with the MovieLens dataset, using several metrics for measuring the change in recommendations. We find that our influence-guided fuzzing heuristics are more effective than randomly generating modifications. For instance, for one recommender system tested, 100 modifications generated with an influence-guided heuristic caused 93% of users to have their top-ranked item removed from their recommendations after retraining, on average. In contrast, on average, 100 modifications generated at random caused the top-ranked item to stop being recommended to only 5% of users.

1.2 Overview

The chapters of this thesis are adapted from two conference papers by the author and colleagues. Chapter 2 provides background on recommender systems. Chapter 3 is adapted from *Characteristic Properties of Recommendation Systems* [54] (currently under submission), and presents our approach to characterizing the behavior of recommender systems with property templates. Chapter 4 is adapted from *Influence-Guided Fuzzing for Testing the Stability of Recommender Systems* [55] (under review at the time of this writing), and presents our influence-guided fuzzing approach for assessing the stability of recommender systems. Finally, we present future work in Chapter 5.

Chapter 2

Background and Related Work

In this chapter, we present a brief background on recommender systems, which are the target application for this work. We then present prior work in the area of recommender system evaluation. We especially look at properties proposed in the literature that extend beyond accuracy metrics to evaluate the usefulness of recommendations, and we discuss how our approach is novel in its broad and formal study of recommender properties. We then present related work on evaluating the robustness and stability of recommender systems. Finally, we discuss background and related work on testing the robustness of software, especially fuzz testing.

2.1 Recommender Systems

Recommender systems are generally distinguished by the method they use to calculate recommendations [3, 7, 42]. The two main distinctions are collaborative filtering recommendation systems and content-based recommendation. Collaborative filtering is based on analyses of user preferences and behavior, generally recommending items that similar users have preferred in the past. Content-based recommendation uses information about items to recommend items that are similar to those a user has previously liked. Recommender systems that combine these approaches are known as hybrid recommender systems.

Recommender systems can also be broken down into memory-based and model-based systems [10]. Memory-based recommender systems compute recommendations directly from the rating data, while model-based recommender systems compute a representative model that can be used to predict user preferences and produce recommendations.

One of the earliest recommender systems was the collaborative filtering system, *Tapestry*, for mailing lists [20]. Since then, many automated collaborative filtering algorithms have been introduced. Some of the most common collaborative filtering algorithms use a *k nearest neighbors (kNN)* approach, in which the nearest neighbors to a user or item are used to predict preferences [51]. For example, Herlocker et al. [27] present a user-based collaborative filtering method in which similar users are computed based on users' ratings on common items, and the most similar users are used to compute recommendations for a target user. Sarwar et al. [50] present a model-based collaborative filtering algorithm which computes a model of the similarity between items based on their rating vectors and that item similarity model for predicting user preferences.

Another common approach to collaborative filtering algorithms is *matrix factorization*, which attempts to reduce the set of ratings to a low dimensional set of latent factors [31]. One of the earliest approaches to matrix factorization in recommender systems used Singular Value Decomposition to reduce the dimensionality of the ratings [49]. Luo et al. [37] present a matrix factorization approach for an incremental recommender system that can be trained on new data as it is received, without retraining the entire model.

Content-based systems recommend items similar to those for which a user has expressed a positive preference [8, 42]. Unfortunately pure content-based recommendation suffers from several drawbacks [7], namely: the tendency to recommend the same types of items to a user; and the limited data from which to infer preferences due to the system only using the ratings of the active user. Because of these drawbacks, pure content-based recommender systems are rare, and are more commonly combined with collaborative filtering based approaches in hybrid recommender systems. For instance, Kula [32] combines content-based and collaborative filtering in the LightFM algorithm. Soboroff and Nicholas [56] use Latent Semantic Indexing to create user profiles based on document content, which can then be used for collaborative recommendation.

Bobadilla et al. [8] provide a more comprehensive survey of the space of recommender systems. In addition to the characterizations defined above, they discuss context-aware recommender systems, which can take into consideration contextual information such as time or location [4]. They also survey the use social information—information about the network of social relationships—in recommender systems. For instance Woerndl and Groh [58] use social network friendships to build neighborhoods of users for collaborative filtering.

2.2 Evaluating Recommender Systems

Precision and recall, popular metrics from the information retrieval community, are commonly used in the evaluation of recommender systems [11, 48, 62]. Precision represents the probability that a recommended item is relevant, while recall represents the proportion of relevant items that are recommended. More specialized metrics, such as the Normalized Discounted Cumulative Gain (NDCG), adjust the weight of items in the ranked list as their position increases [29]. Other metrics, like the Normalized Distance-based Performance Measure (NDPM) [60] and rank correlation measures such as Spearmans ρ or Kendalls τ [28], assess the accuracy by comparing the ordering of ranked items to a known ordering.

Herlocker et. al [28] contend that, in addition to accuracy, recommender systems must provide usefulness, and discuss other measures related to *coverage*, *confidence*, *learning rate*, and *novelty.* McNee et al. [38] also call for new metrics to better measure the quality of recommendations from the perspective of a user, as well as techniques to understand the differences between recommender algorithms. Researchers have also highlighted the limitations of summary statistics like precision and recall and have called for better and alternative characterizations [25]. Our work directly addresses these calls.

Shani and Gunawardana [53] identify 14 such metrics. Among those, we focus on five that can be automatically computed with standard training data and ratings: *prediction accuracy*, *coverage*, *novelty*, *diversity*, and *adaptivity.*

As discussed, accuracy can be captured by a variety of metrics, but the most common approach is to compute precision and recall relative to a set of ideal results. Coverage can be captured by a range of measures, a common metric being the number or proportion of items that a recommender system can recommend [17, 22, 27, 53]. However, coverage can also be measured as "the proportion of users . . . for which the system can recommend items" [53].

Novelty is "the extent to which users receive new and interesting recommendations" [43]. This property also admits various approaches for measurement, such as item popularity-based measures [53, 57] or those based on the distance between an item and a "context of experience" [57].

Diversity captures the dissimilarity of recommendations across the dataset, such as metrics "based on . . . distance between item pairs" [53]. In addition to intra-list similarity, which measures the similarity of items in a recommendation list by using item characteristics, such as genre [62], diversity may also refer to the personalization, or uniqueness, of users' recommendation lists [61]. Adomavicius and Kwon [2] consider the total number of distinct items to be a form of aggregate diversity of a recommender system.

Adaptivity captures the notion that recommendations change with the dataset, for instance "when users rate an item, they expect the set of recommendations to change" [53].

In prior work, researchers either take a broad view (e.g., [53]) and present informal descriptions of recommender properties or they take a narrow view, formalizing a metric to capture a specific interpretation of a property. In our work, we undertake a broad and formal study of characteristic properties of recommender systems. More specifically, we define and systematically explore a space of properties defined by instantiating a set of logical property templates that build on a general model of a recommender system. The property templates formally describe a space of properties for a given recommender system that includes many of the metrics described in the related work.

2.3 Robustness of Recommender Systems

O'Mahony et al. [41] describe two aspects of robustness: *accuracy* and *stability*. *Accuracy*, in regard to robustness, is a measure of the recommendation quality after changes are made to the dataset, while *stability* is a measure of how different the recommendations are after a change is made. Gunawardana and Shani [23] consider robustness to be "the stability of the recommendation in the presence of fake information".

Adomavicius and Zhang [5, 6] take a slightly different view of robustness and stability, defining stability as the consistency of recommendations over some period of time under the assumption that any new ratings added to the dataset completely agree with the prior recommendations. Using this definition of stability, they conclude that model-based algorithms are generally more consistent than memory-based collaborative filtering algorithms. They also find that the stability of a recommender system does not necessarily correlate with the accuracy of the system.

In prior work, recommender system robustness is commonly evaluated in the context of attacking recommender systems. Shilling attacks, or profile-injection attacks, add user profiles to a recommender system with crafted sets of item ratings in order to increase or decrease the position of some item in the recommendations of all users [34, 41]. O'Mahony et al. [41] analyze the accuracy and stability of collaborative filtering based recommender systems against profile injection attacks. Lam and Riedl [34] evaluate the robustness of two collaborative filtering algorithms against shilling attacks and conclude that item-based algorithms are more robust than those that are user-based. However, Mobasher et al. [40] show that item-based algorithms are vulnerable to different types of profile injection attacks and argue that hybrid recommender systems offer a higher degree of robustness. Gunes et al. [24] provide a comprehensive survey on shilling attacks against recommender systems.

Whereas shilling attacks add new user profiles to promote or demote targeted items, the approach we introduce in Chapter 4 is different in that it generates ratings for existing users and items to provide a general assessment of the stability of a recommender system. Additionally, because shilling attacks add new users to a system, they require adding many ratings for each of those new users. For example Lam and Riedl [34] introduce between 25 and 100 users with 3404 ratings each to a dataset of almost 1,000,000 ratings (a percent change of between 8% and 34%). In contrast, our goal is to identify small sets of modifications (under 1% change) that cause significant change to the recommendations produced by the system.

2.4 Software Robustness and Fuzz Testing

The robustness of a software system generally relates to the dependability or trustworthiness of a system and characterizes the ability of a system to exhibit "acceptable" behavior in the presence of exceptional input [52]. The IEEE standard of software engineering vocabulary defines robustness as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [1].

Duran and Ntafos [13] showed that random testing can be a cost-effective testing approach for many programs. The term fuzz testing—fuzzing—was introduced by Miller et al. [39] to describe the generation of random tests for Unix programs, however the term has expanded to include random testing of software systems in general. Fuzz testing is a common approach to testing the robustness of a software system, where random inputs are cheaply generated to attempt to cause the system to crash or enter an unacceptable state [30]. There are three main levels of fuzz testing: blackbox, whitebox, and greybox. Blackbox fuzz testing requires only an executable program and works by randomly mutating inputs and using those generated inputs to test the program [15]. Whitebox techniques use program analysis to guide input generation [19]. Greybox fuzzing uses information from program instrumentation, such as code coverage, to guide input mutations [9]. Fuzz testing can also leverage additional information for input generation, such as input grammars. For instance, Godefroid et al. [18] perform whitebox fuzzing in combination with a grammar to ensure that generated inputs are valid.

Ours is the first work that we know of that applies fuzzing to the dataset used to train a recommender system for stability assessment purposes. Because the space of dataset modifications is so large, we use influence models inferred from the original dataset and recommendations to effectively guide the generation of modifications to the dataset.

Chapter 3

Characterizing the Quality of Recommender Systems

The standard metrics for evaluating and comparing the quality of recommender systems on the Top-K recommendation task are precision, recall, and derivative metrics such as F-values [48]. In general, if a system S_1 has higher precision and recall than a system S_2, then S_1 is generally considered to be "better" than S_2. Yet, precision and recall are limited in several ways.

First, because they are coarse-grained summary statistics, precision and recall provide little information or intuition to the developer or provider of a recommender system about the quality and usefulness of the recommendations the system provides. For instance, in this work, we show that a known hybrid recommendation algorithm has precision and recall superior to a known model-based collaborative filtering algorithm when evaluated on the MovieLens dataset, yet, the hybrid algorithm achieves those gains by providing excessively conservative recommendations. The model-based collaborative filtering algorithm recommends 3.5 times more unique items and is able to produce a unique ranked list for almost every user. Such insights are lost with coarse summary statistics like precision and recall.

Second, when the number of known relevant items is very small relative to the number of items available to be recommended, recommender systems are likely to exhibit extremely low precision and recall values. This is a common occurrence for modern recommendation systems that must provide tens of recommendations from a set of hundreds of thousands or even millions of items, and which have sparse ratings. Users may have as few as one rating in some datasets. In this environment, a system developer will have little intuition as to whether a precision of 0.001% should be considered supremely satisfying or deeply troubling.

Third, not only do precision and recall values for an individual system convey little intrinsic intuition about the system, conclusions about the relative superiority of one system over other systems based on precision and recall values can be highly misleading if the recommendations provided by the putatively superior system are anomalous or counterintuitive.

We are not the first to discuss these statistics' limitations. Researchers increasingly have been highlighting the limitations of summary statistics like precision and recall and have called for better and alternative characterizations [25]. Furthermore, the need for better characterizations can be seen as an instance of the broader problems of *explainability* and *human interpretability* of AI systems (e.g., see Letham et al. [36] and Kulesza et al. [33]). However, while the limitations of such summary statistics are well known [28], there has been little success in defining general properties that overcome these limitations.

Our contributions in this work are:

- We delineate an approach for defining properties of recommender systems that can provide more intuitive characterizations of the nature and quality of the recommendations produced by a recommender system. More specifically, we develop logical templates that define a space of properties for characterizing the quality of a recommendation algorithm with respect to a given dataset in terms of the relations between the algorithm's inputs (user, items, ratings) and its outputs (rankings). We also discuss how these properties relate to existing properties in the literature, namely, *coverage*, *diversity*, *novelty*, and *adaptivity*.

- We study how property instantiations vary across five algorithms that we applied to the MovieLens and Jester datasets, by formally defining the space of such properties through a series of templates, and instantiating a subset of them. We demonstrate that the instantiated templates are able to reveal a variety of inconsistencies in recommendation behavior that are at odds with the relative performance of the algorithms as indicated by their precision and recall.

- We also investigate how properties defined using our approach are affected as the dataset evolves, providing insights into what algorithms and properties are more robust in the presence of an evolving dataset.

3.1 Defining Recommender Systems

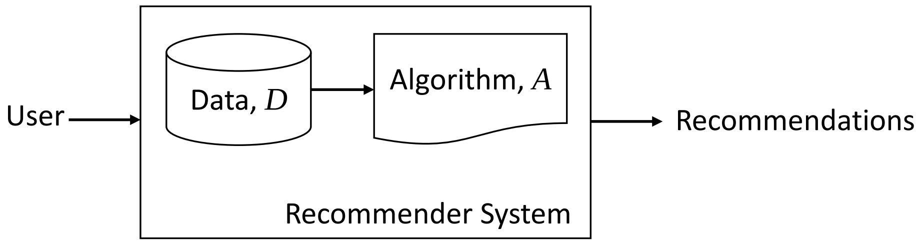

Figure 3.1: Model of a recommender system.

We begin by defining a model for recommender systems. A recommender system consists of an algorithm, A and a dataset, D, as shown in Figure 3.1. The recommender algorithm uses the dataset to compute a model of user preferences. This model can then be used to compute recommendations for users of the system.

A class diagram of our recommender system model is shown in Figure 3.2. This model is composed of users, items, and attributes, and two types of relationship

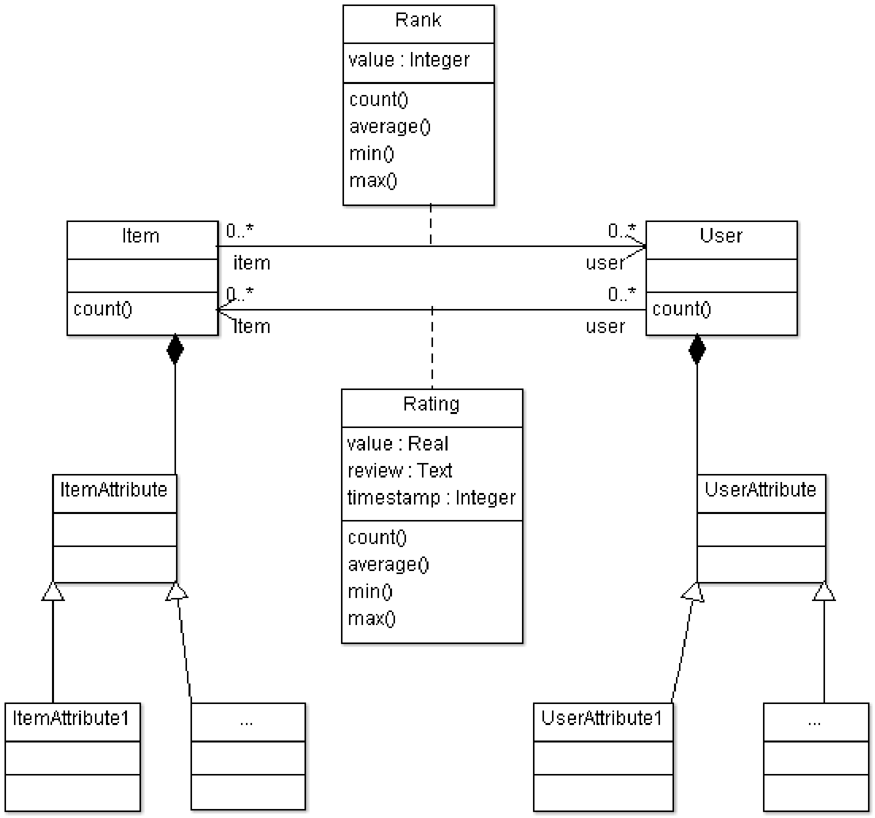

Figure 3.2: Class diagram of a recommender system.

between users and items: ranks, which are relationships between items and users produced by the recommender system; and ratings, which are relationships between users and items defined by the dataset.

Let $\mathcal{U}$ be a finite set of users, $\mathcal{I}$ a finite set of Items, and $\mathcal{R}$ a possibly infinite, ordered set of rating values. Without loss of generality, we assume that $\mathcal{R} \subset \mathbb{Z}_{>0}$ (typically within some interval $[l, h]$).

A dataset D defines the partial function $\mathit{rating}\colon \mathcal{U} \times \mathcal{I} \to \mathcal{R}$ that captures how users rate items, with $\mathit{rating}(u, i) = \perp$ if u has not rated i. In addition to defining rating, D characterizes each $u \in \mathcal{U}$ by attributes drawn from the set $\mathcal{A}_{\mathcal{U}}$, and each $i \in \mathcal{I}$ by attributes drawn from the set $\mathcal{A}_{\mathcal{I}}$ through functions $\mathit{attr}\colon \mathcal{U} \to 2^{A_U}$ and $\mathit{attr}\colon \mathcal{I} \to 2^{A_I}$, respectively. A user or item may be characterized by multiple attributes.

Given D and a parameter k for computing top-k rankings, a recommendation algorithm Q computes a partial function $rank_{k,Q} \colon \mathcal{I} \times \mathcal{U} \rightarrow [1, k]$ for $k \leq |\mathcal{I}|$. $rank_{k,Q}(i, u) = \perp$ if item i is not ranked for user u.

For every $u \in \mathcal{U}$ there are a number of ranked items, $k_u \leq k$, and the projection of rank onto users, $rank_{k,Q}(u) \colon \mathcal{I}_{k_u} \rightarrow [1, k_u]$, is a bijection, where $\mathcal{I}_{k_u} \subseteq \mathcal{I}$, $\#\mathcal{I}_{k_u} = k_u$, and the function # counts the number of elements in the set.

In what follows, we drop the subscript Q from $rank_{k,Q}$ since Q is typically apparent from the context, and we drop the subscript k since k is typically a parameter to Q.

3.2 Recommender System Properties

We seek to provide characterizations of Q for a given D reflecting desirable properties of recommenders that have been identified by the research community.

Metrics for these properties can come in many different forms, and we cannot possibly consider the space of all properties exhaustively. We do, however, want to be able to explore that space systematically and efficiently. In this work, we selectively sample the space by identifying metrics that capture each of the four property categories highlighted in related work: coverage, diversity, novelty, and adaptivity, and then generalize those metrics to property templates that consider a broader set of related metrics. Our approach enables the specification of relatively simple logical property templates, based on users, items, ratings, and rankings, that can be instantiated to capture a broader set of usefulness properties about a recommender system.

A *property template* consists of a logical formula defined over variables that capture features of computed recommendations (e.g, the rank of an item in a recommendation list), and operators that compute derived measures of recommendations (e.g., the maximum or minimum rating of a recommended item). If one instantiates a template by defining each of these parameters, then one can evaluate the truth of the resulting formula over D and *rank*. In the study in Chapter 3.4, we compile partially instantiated templates into computations that, when evaluated over D and *rank*, compute the remaining template parameters.

Tables 3.1, 3.2, and 3.3 contain 12 template formulae organized according to the foregoing property categories. Table 3.1 contains three property template formula that relate to the category of coverage metrics. Table 3.2 contains five property template formulae related to diversity. Table 3.3 contains four template formulae, one of which relates to both coverage and diversity, one that relates to both novelty and adaptivity, one that relates to only novelty, and one that relates to only adaptivity. For each template, the table provides the name, the parameters to be computed, the template definition, and a sample instantiation presented in natural language. For templates that require the parameter f or the parameter $\sim$, f and $\sim$ are functions, where $f \in \{\min, \text{median}, \max, \#, \text{avg}\}$ and $\sim \in \{\leq, =, \geq\}$.

Table 3.1: Property Templates Related to Coverage

Name	**Parameters**	**Property Template**	**Example Instantiated Property**
Coverage			
Number of items per Ranked list (NRANKS)	$k, n\colon \mathbb{Z}_{>0}$	$f\{\#\{i \mid i \in \mathcal{I} \wedge rank(i,u) \neq \perp\} \mid u \in \mathcal{U}\} \sim n$	The minimum (f) number of items ranked in the top 10 (k) for any user is greater than or equal ($\sim$) to 10 (n).
Number of Recommended Items (NRI)	$k, n\colon \mathbb{Z}_{>0}$	$\#\{i \mid i \in \mathcal{I} \wedge \exists u \in \mathcal{U} : rank(i,u) \neq \perp\} \geq n$	There are at least 2791 (n) items in at least one top 10 (k) ranking.
Number of Top Ranked Items (NTRI)	$n\colon \mathbb{Z}_{>0}$	$\#\{i \mid i \in \mathcal{I} \wedge \exists u \in \mathcal{U} : rank(i,u) = 1\} \geq n$	There are 628 (n) items with a rank of 1 in at least one ranked list.

For example, in the last row of Table 3.3, the template property "Number of Ratings of items in the Top-k (NRTK)" characterizes the top recommendations in

Table 3.2: Property Templates Related to Diversity

Name	**Parameters**	**Property Template**	**Example Instantiated Property**
Diversity			
Number of Ranked Lists with Multiple Users (NRLMU)	$k, n\colon \mathbb{Z}_{>0}$	$\#\{\{(i, rank(i,u)) \mid i \in \mathcal{I} \land rank(i,u) \neq \bot\} \mid u \in \mathcal{U} \land \exists u' \in \mathcal{U}\colon \forall i \in \mathcal{I}\colon u' \neq u \land rank(i,u) = rank(i,u')\} \leq n$	No more than 0 (n) top 10 (k) ranked lists are common to more than 1 user.
Number of Ranked Lists (NRL)	$k, n\colon \mathbb{Z}_{>0}$	$\#\{\{(i, rank(i,u)) \mid i \in \mathcal{I} \land rank(i,u) \neq k\} \mid u \in \mathcal{U}\} \geq n$	There are at least 138493 (n) unique lists of 10 (k) ranked items.
Number of Ranked Sets (NRS)	$k, n\colon \mathbb{Z}_{>0}$	$\#\{\{i \mid i \in \mathcal{I} \land rank(i,u) \leq k\} \mid u \in \mathcal{U}\} \geq n$	There are at least 138417 (n) unique ranked sets of 10 (k) items.
Number of Users Per Ranked List (NUPRL)	$k, n\colon \mathbb{Z}_{>0}$	$f\{\#\{u' \mid u' \in \mathcal{U} \land \forall i \in \mathcal{I}\colon rank(i,u') = rank(i,u)\} \mid u \in \mathcal{U}\} \sim n$	The maximum (f) number of users to which a top 10 (k) ranked list is recommended is less than or equal ($\sim$) to 1 (n).
Number of Users to which an Item is Recommended (NUIR)	$k, n\colon \mathbb{Z}_{>0}$	$f\{\#\{u \mid u \in \mathcal{U} \land rank(i,u) \leq k\} \mid i \in \mathcal{I}\} \sim n$	The maximum (f) number of times an item is ranked in the top 10 (k) is greater than or equal ($\sim$) to 48857 (n).

Table 3.3: Property Templates Related to Novelty, Adaptivity, and Coverage and Diversity

Name	**Parameters**	**Property Template**	**Example Instantiated Property**
Novelty			
Item Attribute Order by Number of Recommendations (IANRLCMP)	$k \in \mathbb{Z}_{>0}$ $a_1, a_2 \in \mathcal{A}_{\mathcal{I}}$	$\#\{i \mid i \in \mathcal{I} \land a_1 \in attr(i) \land \exists u \in \mathcal{U}\colon rank(i,u) \neq \bot\} > \#\{i \mid i \in \mathcal{I} \land a_2 \in attr(i) \land \exists u \in \mathcal{U}\colon rank(i,u) \neq \bot\}$	Drama (a_1) movies are recommended in the top 10 (k) more often than IMAX (a_2) movies.
Adaptivity			
Average Rating of items in the Top-k (ARTK)	$k, r\colon \mathbb{Z}_{>0}$	$f\{avg\{rating(u,i) \mid u \in \mathcal{U}\} \mid i \in \mathcal{I} \land \exists u \in \mathcal{U}\colon rank(i,u) \neq \bot\} \sim r$	The minimum (f) average rating for an item in the top 10 (k) is less than or equal ($\sim$) to 1.75 (r).
Coverage and Diversity			
Item-Attributes Never Recommended (IANR)	$k \in \mathbb{Z}_{>0}$ $a \in \mathcal{A}_{\mathcal{I}}$	$\forall i \in \mathcal{I} : a \in attr(i) \implies \forall u \in \mathcal{U} : rank(i,u) = \bot$	IMAX (a) movies are never recommended in any top 10 (k) ranked list.
Novelty and Adaptivity			
Number of Ratings of items in the Top-k (NRTK)	$k, n\colon \mathbb{Z}_{>0}$	$f\{\#\{rating(u,i) \mid u \in \mathcal{U}\} \mid i \in \mathcal{I} \land \exists u \in \mathcal{U}\colon rank(i,u) \neq \bot\} \sim n$	The minimum (f) number of ratings for an item in the top 10 (k) is less than or equal ($\sim$) to 2 (n).

terms of the number of ratings they have. This template is parameterized by k, the number of items considered in the ranking; n, the number of ratings per item; f, the operator used to aggregate rating counts across D; and $\sim$, a relational comparison used in conjunction with f. The sample instantiation of this property sets k to 10, f

to *minimum*, $\sim$ to $\leq$ and n to 2. In our study, we partially instantiate the property with k, f, and $\sim$ and then compute the greatest value of n satisfying the resulting formula.

Each of these template maps onto at least one of the recommender property categories of Shani and Gunawardana [53]. Coverage is captured by NRI, NTRI (coverage of items), NRANKS (coverage of users), and IANR (coverage of item attributes). Diversity is captured by IANR, NRLMU, NRL, NRS, NUPRL, and NUIR (inter-user diversity). Novelty is captured by NRTK (unpopular items may be novel) and IANRLCMP (rarely recommended attributes may be novel). Adaptivity is captured in ARTK (adaptivity to ratings) and NRTK (adaptivity to popularity). As we shall see, the satisfaction of these template formulae, and the recommender properties they capture, vary across recommender algorithms.

Properties versus Metrics. It is worth highlighting that our approach is novel in the treatment of usefulness properties of recommenders as logical properties, rather than as metrics. While metric values are singular and coarse in their attempt to capture a specific property of interest, our approach computes property instantiations that represent a space of possible values for the property. For instance, if the computed instantiation of ARTK $(min, \leq)$ is $n = 1.75$, when $k = 10$, then the property also holds for all values of n less than 1.75, as well as for values of k less than 10. The properties we have identified in this work can be used to capture common definitions of usefulness properties, such as those identified above. Additionally, our approach provides a general model for defining property templates that can capture novel dimensions of these usefulness properties, such as ARTK and NRTK, which provide novel characterizations of a recommender system that are not captured by currently defined metrics. Our IANR property captures the ability of the recommender system to recommend all item attributes, and isn't captured by current metrics. This

```
universe
  .ranks()
  .filter(lambda rank: rank.value <= k)
  .items()
  .forall(lambda item:
    item.ratings().apply('average') >= r)
```

Figure 3.3: Sample Implementation of the ARTK Property Computation in Python.

```
n = universe
        .ranks()
        .filter(lambda rank: rank.value <= k)
        .items()
        .apply('count')
```

Figure 3.4: Sample Implementation of the NRI Property Computation in Python.

property relates to both coverage — it characterizes the ability of the recommender system to cover all item attributes — as well as diversity — how diverse are the item attributes that are recommended.

3.3 Property Instantiation Implementation

We implemented a system that, for a given Q and D, computes the parameter values for a subset of the properties defined by the templates. The system, implemented in Python, consumes items, users, ratings, and rankings, inserting them into sets. It makes use of predefined lambda functions for filtering, aggregating, and iterating on the sets, computing the parameters for which a target property would hold. Examples for five properties are shown in Figures 3.3, 3.4, 3.5, 3.6, and 3.7. In the implemented system, `universe` represents all knowledge of the recommender system being evaluated.

Figure 3.3 presents the computation for the ARTK property with f parameterized

```
universe
  .users()
  .forall(lambda user: (user.ranks()
    .filter(lambda rank: rank.value <= k)
    .apply('count')) >= n)
```

Figure 3.5: Sample Implementation of the NRANKS Property Computation in Python.

```
n = universe
    .ranks()
    .filter(lambda rank: rank.value <= k)
    .apply('max_count', key=lambda rank: rank.item)
```

Figure 3.6: Sample Implementation of the NUIR Property Computation in Python.

```
n = universe
    .users()
    .apply('count_unique', key=lambda user: user.ranks()
           .filter(lambda rank: rank.value <= k)
           .apply('select',
              key=lambda rank: (rank.item, rank.value)))
```

Figure 3.7: Sample Implementation of the NRL Property Computation in Python.

as *min* and $\sim$ parameterized as $\geq$. Starting from the top, this code will find a rating threshold r such that the average rating for all items ranked in the top k is at least equal to to r.

Similar to ARTK, the implementation for the NRI property, presented in Figure 3.4, begins by applying a filter to ranks to select items in the top k, then collecting the items within those ranks, and aggregating them with the function *count*. This has the effect of counting the number of unique items that appear in some top-k ranking.

In contrast to ARTK and NRI, the implementation of the NRANKS property iterates over the set of users and checks whether the number of ranked items for all users is greater than n, as shown in Figure 3.5. This implementation partially instantiates the property with f parameterized as min and $\sim$ parameterized as $\geq$. Starting from a value of k, this code will find a value for n such that the number of ranked items for each user is at least n.

The implementation for the NUIR property also iterates over the rankings, as shown in Figure 3.6. However, this property uses a specially defined `max_count` function that counts the number of times each item appears and returns the highest count value. This has the effect of computing a value of n for the NUIR property parameterized with f as max and $\sim$ as $=$.

Similar to NRANKS, the implementation of the NRL property iterates over the users in a dataset, as shown in Figure 3.7. We then apply the `unique_count` function that counts the number of unique objects in the set. We check whether two objects are unique by comparing their top-k ranked lists, which we build by selecting the item and value from the ranks. This code computes a value, n, equal to the number of unique ranked lists produced by the recommender system.

3.4 Study

We carried out a study to explore the value of our properties in differentiating existing algorithms and revealing unexpected behaviors that were not apparent with standard prediction accuracy metrics, and we also investigate how these properties are affected by changes in the dataset. More specifically, we attempt to answer the following questions:

RQ1: Can the properties offer insights about Q that go beyond what precision and recall already offer?

RQ2: How robust are the properties instantiated for Q in the presence of an evolving training set D?

3.4.1 Study Design

3.4.1.1 Recommendation Algorithms

For this study we selected five different recommendation algorithms. The algorithms were chosen to represent a variety of recommender techniques ranging from memory-based to model-based, and from content-based to collaborative filtering based. An additional criterion that guided our algorithm selection was that a candidate algorithm had to work with the MovieLens dataset [26], either because it was a part of the Lenskit framework or because its adaptation to that framework required only minor data wrangling. We describe each of the five chosen algorithms below.

User-User. The User-User algorithm is a memory-based collaborative filtering algorithm introduced by the GroupLens project [46]. The algorithm computes user similarity scores based on user rating vectors. In this work, we use the implementation of User-User provided by the Lenskit framework [14], which computes user similarity using vector cosine similarity [10].

Item-Item. The Item-Item algorithm is a model-based collaborative filtering algorithm. In this algorithm, similarity scores are precomputed for all pairs of item rating vectors. The predicted value of an item is estimated by aggregating the ratings of the most similar items [12, 50]. We use the Item-Item implementation provided by the Lenskit framework [14].

FunkSVD. FunkSVD is a model-based collaborative filtering algorithm that uses stochastic gradient descent to learn a matrix factorization [16]. In this work, we use the implementation of FunkSVD provided by the Lenskit framework [14], which learns 25 latent features.

Slope One. The Slope One algorithm is a model-based collaborative filtering algorithm that uses the average deviation of the ratings between items a user has rated and the item being scored [35]. We use the implementation of Slope One provided by the Lenskit framework [14].

LightFM. LightFM is a model-based hybrid recommender algorithm that combines content-based and collaborative filtering based recommendation. It uses both ratings and item attributes to build a recommender model [32]. In our study, we use the Python implementation of the algorithm provided by the algorithm's author[1]. This is the only algorithm that explicitly uses item attributes when building a recommender model. Using the MovieLens dataset, we provide the genres as item attributes.

3.4.1.2 Datasets

MovieLens 20M Dataset. We use the dataset released in 2016 from the MovieLens recommendation system. The dataset is available as a group of comma separated files, containing over 20 million ratings collected from over 130,000 users over a period of 20 years on 27,000 movies. Ratings can have ten discrete values, from 0.5 to 5.0 with a step size of 0.5. More details about the data collection process and the dataset itself are available at `https://grouplens.org/datasets/movielens/20m/` [26].

Jester Dataset. We also use the Jester Dataset collected between November 2006 and May 2009 [21]. The dataset is available as a group of tab separated files, containing over 1.7 million ratings from 59,132 users on 150 jokes. Ratings in this dataset are on a continuous scale from -10.0 to 10.0. More details about this dataset are available at `http://eigentaste.berkeley.edu/dataset/`.

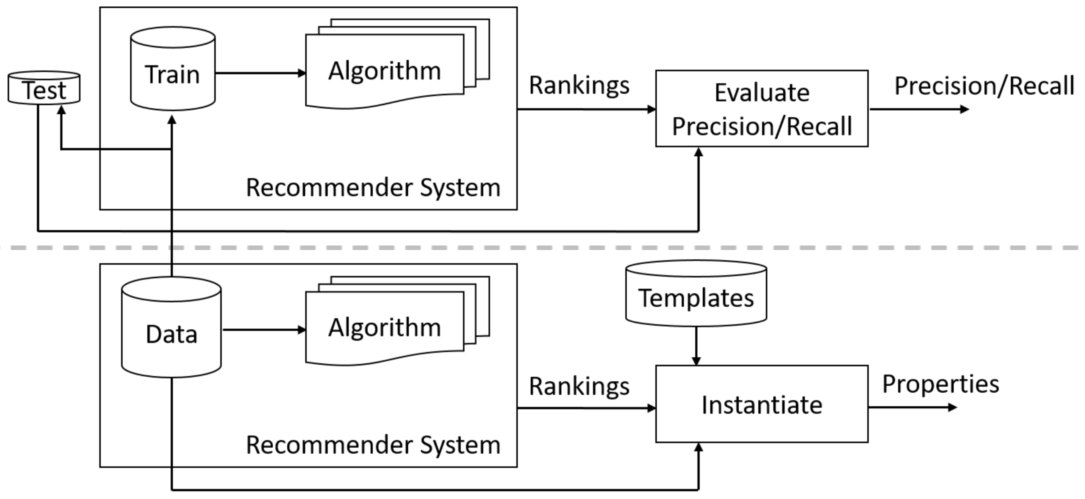

Figure 3.8: Design of the Study.

3.4.1.3 Design

The overall design of our study is shown in Figure 3.8. We build two recommender systems per algorithm and dataset. One recommender system splits the data into training and test sets and performs a traditional precision and recall evaluation by training the recommender on the training set, and using the test set to evaluate the metric. We perform a five-fold cross-validation, partitioned on the users, and using an 80-20 split on users' rated items. We consider items in the test set that are rated by some user to be relevant for that user. We calculate precision and recall for each user as $\frac{\#(TopK \cap RelevantItems)}{\#TopK}$ and $\frac{\#(TopK \cap RelevantItems)}{\#RelevantItems}$, respectively. We report the average precision and recall across all users. The second recommender uses the full dataset and is used to instantiate the properties according to the templates. We use all algorithm and data combinations except for the LightFM recommender algorithm with the Jester dataset because it does not provide item attributes, which were required by LightFM.

To answer the second research question, we used the MovieLens dataset. We chose

[1]`https://github.com/lyst/lightfm`

this dataset because it has timestamps associated with each rating, while the Jester dataset does not. To simulate dataset evolution, we sorted all of the ratings by their timestamps, and selected the earliest 50%, 60%, 70%, 80%, 90%, and 100% of the dataset to mimic its evolution while following the same instantiation process as for the first research question. To have consistent datasets, for each subseries, the item set was restricted to contain only movies that were released in or before the year in the latest rating timestamp. Using this method, each subseries is a subset of the next largest subseries and simulates the evolution of the dataset over time.

3.5 Study Results

3.5.1 RQ1: Properties across Algorithms

We start by computing the precision and recall and the instantiated property values using the full dataset. The results for the MovieLens dataset are shown in Table 3.4, and the results for the Jester dataset are shown in Table 3.5. To facilitate the presentation of the results, cells in a row with similar values are similarly colored to indicate comparable algorithm performance according to the row's property. For each property, values are grouped into three groups, based on their similarity. Groups are formed to maximize the distance between the values in each group. For properties with only two distinct values, such as ARTK$(max, \geq)$, we only form two groups. Overall, we observe that the properties are able to reveal many differences across algorithms with similar precision and recall. We illustrate some of those differences by property type.

Similar precision and recall, but different coverage. Using the MovieLens dataset, User-User and Slope One have similar values for precision and recall, but Slope One has higher coverage of the item set, recommending 1580 unique items

Table 3.4: Precision, Recall, and Instantiated Property Values for Each Recommender System using the MovieLens dataset

Category	**Property**	**User-User**	**Item-Item**	**FunkSVD**	**Slope One**	**LightFM**
Accuracy	**Precision**	1.45E-06	1.79E-03	1.40E-03	1.44E-06	5.20E-03
	Recall	6.68E-08	3.41E-04	3.36E-04	3.99E-08	2.13E-03
Coverage	**NRANKS** $(min, \geq)$	0	10	10	10	10
	NRI	465	2791	1798	1580	786
	NTRI	89	628	579	474	163
Diversity	**NRLMU**	3442	0	10740	5465	233
	NRL	132390	138493	47065	126970	138250
	NRS	125588	138417	13180	87842	85784
	NUPRL $(max, \leq)$	90	1	782	42	3
	NUIR $(max, \geq)$	78575	48857	135909	104763	99833
Novelty	**IANRLCMP**	Drama, Comedy, Horror, …	Drama, Comedy, Romance, …	Drama, Documentary, (no genres listed), …	Drama, Documentary, Comedy, …	Adventure, Comedy, Action, …
Adaptivity	**ARTK** $(min, \leq)$	0.5	1.75	1.79	0.5	0.5
	ARTK $(med, =)$	1.50	3.80	3.65	3.75	3.24
	ARTK $(max, \geq)$	5.0	4.83	5.0	5.0	5.0
Coverage & Diversity	**IANR**	$\{IMAX\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{(no\ genres\ listed)\}$
Novelty & Adaptivity	**NRTK** $(min, \leq)$	1	2	1	1	0
	NRTK $(med, =)$	2	6	1756.5	1	71
	NRTK $(max, \geq)$	47	67310	67310	70	66172

while User-User recommends only 465 unique items. User-User also has a lower item-attribute coverage for the MovieLens dataset, never ranking an item with the “IMAX” attribute in any top 10 list, while Slope One recommends every item attribute at least once. Similarly, although LightFM has slightly higher precision and recall values than FunkSVD, the NRI property indicates that it has a lower coverage of the item set and it never ranks an item with the “(no genres listed)” attribute in the top-10 list of any user (as shown by the IANR property), while FunkSVD recommends every attribute at least once.

When using the Jester dataset, User-User and Slope One have similar values for precision and recall, as they did with the MovieLens dataset. However, unlike with MovieLens, User-User has a higher coverage of the item set, recommending 139 unique items while Slope One recommends only 123 unique items. Additionally, User-User recommends 130 unique items as the top item of some ranked list, while Slope One

Table 3.5: Precision, Recall, and Instantiated Property Values for Each Recommender System using the Jester dataset

Category	Property	User-User	Item-Item	FunkSVD	Slope One
Accuracy	**Precision**	0.167	0.166	0.117	0.178
	Recall	0.195	0.177	0.112	0.186
Coverage	**NRANKS** $(min, \geq)$	0	0	0	0
	NRI	139	132	136	123
	NTRI	130	109	134	30
Diversity	**NRLMU**	588	1221	369	4405
	NRL	55093	53766	56543	15698
	NRS	51805	31476	48880	10908
	NUPRL $(max, \leq)$	1585	1451	775	7499
	NUIR $(max, \geq)$	38522	34363	14991	39623
Adaptivity	**ARTK** $(min, \leq)$	-2.75	-2.75	-2.75	-0.70
	ARTK $(med, =)$	1.94	1.97	1.95	2.11
	ARTK $(max, \geq)$	3.71	3.71	3.71	3.71
Novelty & Adaptivity	**NRTK** $(min, \leq)$	166	166	166	166
	NRTK $(med, =)$	9620	9322	9382	9669
	NRTK $(max, \geq)$	57720	25996	54150	25996

has only 30 distinct top-ranked items.

Using either dataset, we see that information about the coverage behavior of a recommender system is complementary to precision and recall. Two recommender systems having similar values of precision and recall does not preclude them from having discrepancies in their coverage of the set of items.

Additionally, we see that instantiated property templates provide insights into recommender system behavior that result from the structure of the dataset. For instance, although User-User and Slope One have similar precision and recall for both the MovieLens and Jester datasets, Slope One has higher coverage than User-User on the MovieLens dataset, but a lower coverage when using the Jester dataset. We conjecture that this disparity in behavior may arise due to the lower number of items in the Jester dataset, as well as the higher density of ratings in the dataset. In Jester, ratings exist for over 20% of user-item pairs, while for MovieLens, only about 0.5% of user-item pairs are rated.

Similar precision and recall, but different diversity. When using the MovieLens dataset, FunkSVD and Item-Item recommenders have similar precision and recall values, but the Item-Item recommender has a higher diversity as seen in the NRLMU and NRL properties. Item-Item provides a unique ranked list for every user, while the FunkSVD recommender only produces 47,065 ranked lists. Additionally, LightFM has slightly higher precision and recall than Item-Item. However, the NRS property indicates that it has a lower inter-user diversity, recommending only 85,784 unique ranked sets, while Item-Item recommends 138,417 ranked sets.

When using the Jester dataset, User-User and Slope One have similar values for precision and recall, but the User-User recommender has a higher diversity, as evidenced by the NRS property. User-User produces 51,805 unique sets of recommendations, while the Slope One recommender recommends only 10,908 unique sets of items.

For both the MovieLens and Jester datasets, at least one property template that captures the diversity of recommendations can differentiate recommender systems with similar precision and recall. In fact, using the similarity groups identified in Tables 3.4 and 3.5, the NRS property is enough to differentiate recommenders with similar precision and recall. This is likely due to our selection of algorithms that represent a variety of recommendation techniques, however it shows that the diversity of recommendations can vary greatly for given values of precision and recall.

Similar precision and recall, but different novelty. FunkSVD makes up for diversity with novelty as it can recommend items with fewer ratings than Item-Item, for the MovieLens dataset. FunkSVD recommends at least one item with a single rating, while Item-Item never recommends an item with fewer than two ratings. The slightly higher precision and recall of LightFM, when compared to FunkSVD and Item-Item, is in part due to its ability to recommend items without any ratings data, which is captured by the property NRTK($min, \leq$). Also, because FunkSVD and Item-Item account only for ratings data, they tend to recommend items with higher average ratings. The ARTK($min, \leq$) property shows that FunkSVD never recommends items with an average rating less than 1.79 and Item-Item never recommends items with an average rating less than 1.75. In contrast, LightFM recommends at least one item with an average rating of 0.5, the lower rating bound for the MovieLens dataset.

In addition to different coverage and diversity, the User-User and Slope One recommenders also have different novelty when using the Jester dataset. Slope One never recommends items with more than 25,996 ratings, while User-User recommends at least one item with 57,720 ratings, as shown by NRTK($max, \geq$). The User-User and Item-Item recommender systems also achieve similar precision and recall on the Jester dataset, however Item-Item produces more novel recommendations, as shown by the NRTK($med, =$) and NRTK($max, \geq$) properties.

Using either dataset, we see that properties that capture the novelty of recommendations offer complementary insights into precision and recall. For instance, the NRTK($min, \leq$) instantiation for LightFM on MovieLens provides insight into its slightly higher precision and recall. Additionally, for both datasets, the NRTK($med, =$) property provided the most discrimination among recommenders with similar precision and recall values.

Similar precision and recall, but different adaptivity. Although they have similar precision and recall on the MovieLens dataset, FunkSVD is more sensitive to the value of an item's ratings than the Item-Item recommender system, as the Item-Item recommender never recommends an item with an average rating greater than 4.83. Additionally, although Item-Item and LightFM have similar precision and recall, they differ in their instantiations for all three ARTK properties (($min, \leq$), ($med, =$), ($max, \geq$)).

While User-User and Slope One have similar precision and recall on the Jester dataset, Slope One is more sensitive to rating values. The ARTK($min, \leq$) property shows that the Slope One recommender never recommends an item with an average rating below -0.70, while User-User recommends at least one item with an average rating value of -2.75, the lowest average rating value in the Jester dataset.

With both the MovieLens and Jester datasets, instantiated property templates that capture the adaptivity of a recommender system can often differentiate recommender systems with similar precision and recall. While the ARTK properties are often able to differentiate recommenders using the MovieLens data, they provide less discriminating power for the Jester dataset, where User-User, Item-Item, and FunkSVD all have similar values. We conjecture that this lack of differentiation is due to the limited number of items that can be recommended in the Jester dataset, with 2982 users rating more than 90% of the available items. Because these users have rated so many items, the items available for recommendation are limited, resulting in restricted adaptivity.

Instantiated property templates provide insights into recommender system behavior that results from the underlying algorithm. While the precision and recall values are similar for User-User and Slope One on both datasets, for both MovieLens and Jester, Slope One was more sensitive to rating values. For the MovieLens dataset, the ARTK($med, =$) property for Slope One is more than double that of User-User, indicating it tends to recommend items with much higher average ratings. Using Jester, both ARTK($min, \leq$) and ARTK($med, =$) are higher for Slope One than User-User. This difference in behavior is likely due to how the recommender scores items. Because Slope One uses an item's average rating deviation from other items to predict the preference of a user for an item, items with high average ratings are more likely to be recommended. User-User on the other hand uses the correlation between users ratings to generate recommendations. Because of this, User-User can recommend items with low ratings if two users rating sets have a negative correlation.

Different precision and recall, but similar instantiated property. While the FunkSVD and Slope One recommenders are not similar in terms of precision and recall, when using the MovieLens dataset, they recommend similar numbers of items in the top 10 (NRI). They also have a strong correlation (Kendall's $\tau = 0.66$) between their ordering of item attributes for the IANRLCMP property. In addition, the Slope One and LightFM recommenders are not similar when comparing precision and recall; however, they recommend similar numbers of ranked sets, and they have similar values for the NUIR property.

With the Jester dataset, the FunkSVD and User-User recommenders are not similar in terms of precision and recall, but they provide similar item coverage, shown by the NRI property. FunkSVD recommends 136 unique items, while User-User recommends 139. They also provide similar levels of diversity, with FunkSVD producing 56,543 unique ranked lists and User-User producing 55,093 unique ranked lists, as shown by NRL.

While precision and recall are sensitive to a small subset of recommendations, instantiated property templates provide a characterization of the overall behavior of a recommender system. For both the MovieLens and Jester datasets, we see that when two recommender systems have different values of precision and recall, they are still often similar for at least one property. In some cases, recommender systems have different precision and recall, but they still have many similar property template instantiations.

Anomalies. Contrary to expectations, we found that, when using the MovieLens dataset, one of the popular algorithm implementations, User-User, does not have full coverage of the set of users as evidenced by the NRANKS property. Under this implementation, users who have 0 variance in their ratings will not receive a ranked list under User-User.

We also found, for the Jester dataset, that none of the algorithms provided full coverage of the set of users (again shown by the NRANKS property). This is because, in this dataset, some users have rated all of the available items, and the recommender algorithms used in this work do not recommend items that have been previously rated.

3.5.2 RQ2: Properties across Evolving Datasets

As mentioned earlier, to evaluate the stability of our properties on an evolving dataset, we created six datasets from the MovieLens dataset consisting of 50%, 60%, 70%, 80%, 90%, and 100% of the original dataset. Using each of the six subsets, we computed precision and recall, and normalized the values for our properties under each algorithm. We then computed the absolute value of the percentage change for each property between pairs of consecutive datasets to quickly assess the degree of adjustment in a property as a function of the evolving dataset.

Table 3.6 shows the maximum of the absolute values of percentage change for each property for a given algorithm, with values under 5% highlighted. Such highlighted values represent what we deem to be robust properties per recommendation system as the dataset evolves, and in the cases with 0% we say those properties are invariants under varying datasets.

As can be seen in Table 3.6, precision and recall are not robust to changes in the dataset and extremely large changes are observed between consecutive subseries. Only for LightFM does the change in precision and recall remain under 50% between all consecutive datasets. For the rest of the recommender systems the percentage of variation is at least in the hundreds. On the other extreme, a property like NRANKS

Table 3.6: Maximum Absolute Value of Percent Change Between Instantiations of a Property as the MovieLens Dataset Evolves

Category	Property	User-User	Item-Item	FunkSVD	Slope One	LightFM
Accuracy	**Precision**	4121.02%	297.16%	963.42%	110269.67%	25.15%
	Recall	4389.77%	456.89%	1494.53%	446349.93%	48.40%
Coverage	**NRANKS** $(min, \leq)$	0.00%	0.00%	0.00%	0.00%	0.00%
	NRI	129.38%	28.34%	56.89%	48.39%	20.15%
	NTRI	102.80%	54.88%	12.24%	47.63%	23.43%
Diversity	**NRLMU**	6541.91%	0.00%	212.43%	532.07%	86.10%
	NRL	2.21%	0.00%	494.22%	7.15%	0.27%
	NRS	18.02%	2.65%	765.80%	25.82%	12.83%
	NUPRL $(max, \geq)$	18.14%	12.27%	1691.55%	191.90%	48.24%
	NUIR $(max, \geq)$	80.16%	80.06%	0.51%	30.66%	6.67%
Adaptivity	**ARTK** $(min, \leq)$	0.00%	25.00%	24.92%	0.00%	100.00%
	ARTK $(med, =)$	75.96%	2.06%	1.45%	7.69%	1.63%
	ARTK $(max, \geq)$	0.00%	1.96%	0.00%	0.00%	16.04%
Novelty & Adaptivity	**NRTK** $(min, \leq)$	13.05%	13.05%	13.05%	13.05%	0.00%
	NRTK $(med, =)$	2035.15%	953.27%	33.87%	761.55%	100.00%
	NRTK $(max, \geq)$	135166.76%	0.00%	0.00%	20585.97%	25.38%

seems to be a true invariant, independent of the algorithm and dataset, as it remains constant for all algorithms, despite changing data. More commonly, properties are robust for a certain category of algorithms. For instance, the ARTK $(max, \geq)$ is robust only for algorithms that exclusively use ratings data. Such algorithms tend to recommend items with higher ratings due to their limited knowledge, while systems that use other information to generate recommendations can cause the property to become unstable. Other properties seem to be too easily falsified. For example, NRTK $(med, =)$ and NUPRL do not seem robust enough for any algorithm. This may indicate that these properties are too closely coupled to the dataset.

From an algorithmic perspective, some algorithms appear much more sensitive than others to the evolving dataset. For instance, the Item-Item algorithm has seven (7) properties that are robust as the dataset evolves, while the Slope One only has three (3) robust properties. For some algorithms, none of the properties in a given category remain robust in the presence of change. For example, none of the diversity properties are invariant for the Slope One algorithm. Such a finding tells us that the amount of diversity in recommendations from Slope One is highly variable, and that a measured value of diversity may change by a large margin as the dataset evolves.

Chapter 4

Testing the Stability of Recommender Systems

For companies that rely on recommender systems, such as Amazon and Netflix, the ability of their system to make good recommendations in the presence of noise, the system *robustness*, can have large impacts on both user experience and profits [41]. In this work we focus on *stability* robustness, which measures how recommendations could change when the system is trained on modified data, regardless of the recommendation quality. Intuitively, a system lacks stability when there exist a set of changes to the dataset that can have a disproportionate effect on the recommendations. The lack of stability is problematic in that it causes recommendation inconsistencies that can lead to loss of user trust [6].

Determining the stability of a system, however, is extremely challenging due in part to the size of the input space that needs to be explored. As an example, the MovieLens dataset we use in this thesis has 943 users, 1682 items, and 100,000 ratings on a scale from one to five. A naive approach that simply adds a single random rating to that dataset would likely overestimate the system stability since finding one addition among the eight million (943 users x 1682 items x five ratings) possible ratings that can cause instability is very unlikely. Similarly, an exhaustive search for instability-inducing modifications seems infeasible, especially when considering multiple modifications (i.e., adding k new ratings to our dataset means exploring a space of at least $10^{(6*k)}$ potential modifications) or when considering datasets that can contain hundreds of millions of data points. Furthermore, even if generating the modifications becomes feasible, the system must be retrained under the modified dataset, which is often a more expensive process.

Our approach to this problem is based on the insight that *the recommender algorithms underlying these systems tend to rely on relationships between aspects of the dataset.* These algorithms try to identify influential users, items, or attributes that can be exploited to compute recommendations for other users or items in the system. For instance, a user-based recommender algorithm assumes that influence between users is important and uses similarity measures between users to determine influence. These influences are encoded by the training process in complex data structures that vary from system to system, are not typically exposed, and even when exposed are hard to interpret as their meaning is not explicit (e.g., a series of large covariance matrices, a neural network). However, since these influence relationships are used to generate recommendations, we can approximate the influence of aspects of the dataset using the recommendations produced by the system and the ratings it was trained on. For instance, we can approximate the influence of a user based on the number of items rated by that user which are recommended to other users.

Based on this insight, we present an approach that leverages inferred models of influence to fuzz datasets used to train recommender systems to assess their stability. As we shall see, this fuzzing approach provides a significantly better quantification of stability to raise developers' awareness about potential robustness concerns. The examples of changes and types of changes to the dataset that cause instability provided by the approach can guide developers in adjusting the parameters of the recommender algorithm or even to swap the algorithm to better align it with the stability requirement criteria for a given dataset. Last, stability results can shape post-deployment data training policies, such as determining when to trigger a retraining cycle or defining procedures to preprocess the data to limit the influence of some aspects or records.

Our overall contributions are:

- We approximate influence relationships learned by recommender systems from the recommendations and training data. These influence models can then be used to guide a search for sets of modifications that cause disproportionate differences in computed recommendations.

- We develop an approach for influence-guided fuzzing of recommender system stability. We define several influence models that infer influence from a dataset and recommendations, as well as several dataset fuzzing heuristics. We implement our approach in a tool for generating modifications sets that are more likely to cause instability.

- We evaluate our approach on several recommender algorithms with the MovieLens dataset, using several metrics for measuring the change in recommendations. We find that our influence-guided fuzzing heuristics are more effective than randomly generating modifications. For instance, for one recommender system tested, 100 modifications generated with an influence-guided heuristic caused 93% of users to have their top-ranked item removed from their recommendations after retraining, on average. In contrast, on average, 100 modifications generated at random caused the top-ranked item to stop being recommended to only 5% of users.

4.1 Approach

In this section we present our approach to assessing the stability of recommender systems by fuzzing the dataset used to train the recommender. A diagram of our approach is shown in Figure 4.1. Our approach infers an influence model from a dataset D and a set of recommendations R. The inferred influence model is then used for fuzzing D to produce a new dataset D'. Using the modified dataset we train a new recommender system and generate recommendations R'. The recommendations R and R' are then compared using a differential oracle to assess the instability of the original recommender system.

To present our approach, we use the recommender system model defined in Section 3.1. We begin by describing how we approximate recommender systems with inferred models of influence. Next, we describe how we can use these models for influence-guided fuzzing of modifications to a dataset. Finally, we define differential oracles for recommender stability, and discuss the assumptions and practical considerations of our approach.

4.1.1 Inferring Influence

Our work makes use of inferred models of influence to guide generation of instability-inducing modifications. The influence of users and items on recommendations has been studied in prior work. Rashid et al. [45] introduce a Hide-One-User approach to computing user influence scores. They introduce the metric $NUPD_{u_i}$ to be the number of users whose recommendations change if user u_i is left out of the dataset. Rashid [44] also examines item influence using a Leave-One-Out approach. Our inferred influence models approximate the influence of aspects in a recommender system based on the dataset and recommendations, without retraining the recommender sys-

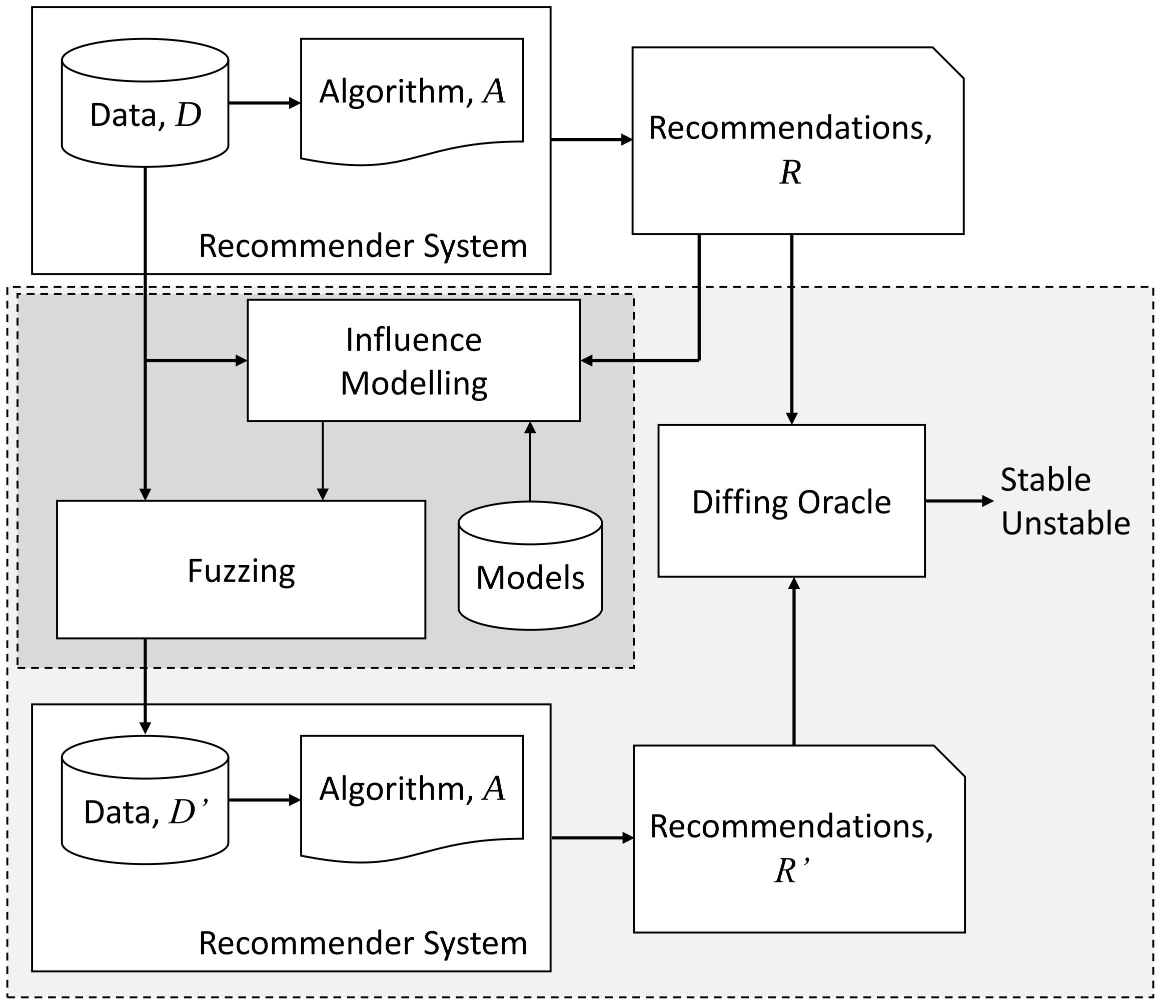

Figure 4.1: Diagram of our approach for generating modified datasets based on inferred influence models.

tem on modified datasets.

Ratings in a dataset implicitly represent a graph structure connecting users based on similarly rated items. Finding influential nodes in this graph can then draw from the field of sociology and the task of identifying important nodes in social networks. One such measure of importance is “centrality”, which can be measured based on the degree of a node, where high degree nodes are considered to be important [59]. “Centrality” can also be based on distances between nodes, where nodes with shorter paths to other nodes are important [59]. Social graph influence measures are similar to our inferred influence models, however they rely strictly on the dataset.

In our approach we define approximations for the influence learned by a recommender system by using functions defined over the dataset used to train a recommender system and the recommendations it produces. We define four influence functions for four types of aspects in a dataset: users, items, ratings, and item attributes.

Recommender systems generate recommendations based on some notion of influence between aspects of the dataset, such as users ($\mathcal{U}$), items ($\mathcal{I}$), or attributes ($\mathcal{A}_\mathcal{I}$ or $\mathcal{A}_\mathcal{U}$). For instance, a user-based algorithm computes users that are influential to a given user, based on the similarity of ratings between pairs of users. Users with high similarity in their ratings are considered more influential to each other than users with low similarity in their ratings. In an item-based recommender algorithm, influence occurs between items, and is computed based on the similarity of ratings between items.

Because influence is used to produce recommendations, we conjecture that we can approximate the influence of various aspects (users, items, ratings, or attributes) of a dataset for a recommender system, based on the data used to train the recommender and the recommendations produced by the system. By observing the relationship

Table 4.1: Influence Model Functions

Name	Definition	Example
`User`	$I^{\mathcal{U}}(u) = \#\{u' \mid rank(i,u') \neq \bot \wedge rating(u,i)\}$	For the User-User recommender algorithm on the MovieLens 100k dataset, user 13 is the most influential user, with an influence score of 858.
`Item`	$I^{\mathcal{I}}(i) = \#\{u \mid rank(i',u) \neq \bot \wedge rating(u,i)\}$	For the Item-Item recommender algorithm on the MovieLens 100k dataset, item 288 is the most influential user, with an influence score of 538.
`Rating`	$I^{\mathcal{R}}(r) = \#\{i \mid rank(i,u) \neq \bot \wedge \lvert r - avg\{rating(u',i)\}\rvert \leq \epsilon\}$	For the FunkSVD recommender algorithm on the MovieLens 100k dataset, rating value 5.0 is the most influential, with an influence score of 12.
`Attribute`	$I^{\mathcal{A}_{\mathcal{I}}}(a) = \#\{(u,i) \mid rank(i,u) \wedge a \in attr(i)\}$	For the LightFM recommender algorithm on the MovieLens 100k dataset, genre "Action" is the most influential, with an influence score of 7029.

between some aspect in the dataset and some aspect in the recommendations, such as whether user u rated an item that is recommended to user u', we can build an approximate model of the overall influence. In general, we approximate influence scores with a function:

$$I^{\mathcal{A}} : \mathcal{A}^n \rightarrow \mathbb{R}, \text{ where } \mathcal{A} \in \{\mathcal{U}, \mathcal{I}, \mathcal{R}, \mathcal{A}_{\mathcal{I}}, \mathcal{A}_{\mathcal{U}}\} \tag{4.1}$$

This function maps aspects of the dataset to a real value representation of the influence of that aspect. In this work, we focus on influence functions over single types of aspects ($n = 1$), however richer forms of influence are possible. Using the computed influence scores, we can approximate the internal influence model of a recommender system as a list of aspects and their associated inferred influence score.

In this work we present four models of influence, which are shown in Table 4.1.

These models of influence capture the four major aspects of a recommender system: users, items, ratings, and attributes. The first column of Table 4.1 gives a short descriptive name to the influence model. The second column provides a definition in terms of the recommender system elements defined earlier. The superscript of the influence function signifies the type of aspect over which this influence applies. The influence function computes a score for an aspect based on the dataset and the rankings produced by the recommender system of interest. We describe each of these models in more detail below. The third column provides a concrete example of the influence evaluated for a single aspect of a recommender system.

User influence measures the impact of a user u on all other users. A user, u, is considered to have impacted another user u' if u has provided a rating for an item appearing in the top-k ranked list of u'. This model of influence captures the intuition that if a user affects the recommendations of many users, then that user is likely influential to the recommender system. For example, for the User-User recommender algorithm and the MovieLens 100k dataset described later, the most influential user is user 13, who has rated items that are recommended to 858 other users.

Item influence measures the impact that an item i has on recommendations, by counting the number of users that have rated item i (and are recommended at least one item). This model of influence captures the intuition that items rated by many users are likely to be more influential. For example, using the Item-Item recommender algorithm and the MovieLens 100k dataset, the most influential item is item 288, which is rated by 538 users.

Attribute influence measures the impact of an item attribute on the recommendation of items with that attribute. An attribute, a, is considered to be impactful if many recommended items have attribute a. This influence model captures the intuition that if many recommended items have a similar attribute, then that attribute must be significant. For example, for the LightFM recommender, the most influential movie genre is "Action" because items with this genre are recommended 7029 times, more than any other genre.

Rating influence measures the impact of rating values on recommendation. A rating value r is considered to be more impactful if more recommended items have an average rating within some value ϵ of r. This model captures the intuition that if many recommended items have similar average rating values, then that rating value is more influential. In other words, items with average ratings near r are more likely to be recommended if r has high influence. For example, for the FunkSVD recommender, the most influential rating value is 5.0 because 12 recommended items (out of 31 distinct recommended items) have a mean rating value of 5.0. The notion of influence for users, items, and attributes is intuitive, but the notion of rating influence is less intuitive. However, we see in Section 4.2 that it can be unusually effective.

4.1.2 Influence-Guided Fuzzing

Using influence models such as those defined above, we can define fuzzing heuristics that produce a set of modifications to the original dataset, and that are likely to cause a recommender system to exhibit unstable behavior.

We allow three basic types of modifications to the dataset: add, remove, and change. *Add* inserts a new rating into the dataset for a user u, item i, and value r if no rating value existed for u and i in the original dataset. *Remove* deletes an existing rating for a user u and item i from the data set. *Change* deletes an existing rating for a user u and item i and inserts a new rating with value r for u and i.

We can define a modification fuzzing heuristic as a function:

$$M : I \times D \times \mathbb{Z} \to 2^{\mathcal{M}} \tag{4.2}$$

$\mathcal{M} \in \{A, R, C\}$, where A is a set of additions, R a set of removals, and C is a set of changes. As per Equation 4.2, this function takes an influence function, a dataset, and a number of modifications to be made, and outputs a set of additions, a set of removals, and a set of changes, or some subset of these. In this work we consider only heuristics that create modification sets of a single type.

We define a small sample of possible fuzzing heuristics, shown in Table 4.2. These heuristics were chosen by keeping the modification type constant (as the Add modification type) and varying the influence type, and by keeping the influence type constant (as user influence) and varying the type of modification, as shown in Table 4.3. This is not a complete listing of possible heuristics but is designed to cover a variety of influence models and modification types. We discuss the intuition for each of these heuristics below. Each row of Table 4.2 is a fuzzing heuristic. The first column of each row provides a short name which we use as an identifier. The second column provides a description of the heuristic with the first word of the description being the modification type, and the third column lists the influence model that is used.

Table 4.3 shows the treatments evaluated in our study, where rows group the modification fuzzing heuristics by the type of influence used, and columns group the heuristics by the type of modifications they produce. This selection of treatments lets us explore random fuzzing vs influence-guided fuzzing by comparing the effectiveness of the heuristics at generating instability-inducing Add modifications. It also allows us to explore how the type of modification (Add, Change, Remove) affects the ability of an influence-guided fuzzing heuristic to generate instability-inducing modifications.

The influence-guided fuzzing heuristics defined here are given three letter names based on how they operate. The first letter is based on the type of modification they produce, where A is for add, C is for change, and R is for remove modifications.

Table 4.2: Fuzzing Heuristics

Name	Description	Influence
AMU	**Add** a rating with random value to a random item for the most influential user.	User
RMU	**Remove** a rating from a random item for the most influential user.	User
CMU	**Change** the rating of a random item to the low value for the most influential user.	User
ALI	**Add** a rating with a random value to the least influential item.	Item
AMR	**Add** a random rating value to an item with an average rating near the most influential average rating.	Rating
AMA	**Add** a rating with the low value to a random user for an item with the most attribute influence.	Attribute

Table 4.3: Treatments Studied

	Modification Type		
Influence	Add	Change	Remove
Random	ARAND	CBRAND CTRAND	RRAND
User	AMU	CMU	RMU
Item	ALI		
Rating	AMR		
Attribute	AMA		

The second letter is the area of the influence model they select aspects from. An M means that the heuristic chooses the most influential aspect and an L means that it chooses the least influential. The third letter specifies the type of influence used by the heuristic. User influence is specified by a U, item influence by an I, attribute influence by an A, and rating influence by an R.

AMU. The AMU fuzzing heuristic adds random ratings to the user with the highest influence score and is defined as:

$$\texttt{AMU} : I^{\mathcal{U}} \times D \times \mathbb{Z} \rightarrow 2^{A} \tag{4.3}$$

Items are selected from the set of items not yet rated by that user and both items and rating values are selected uniformly at random. The intuition behind this heuristic is that by adding ratings to the highest influence user, we may be able to cause a new item to be recommended to many other users, or cause a previously recommended item to stop being recommended for many users.

`RMU`. The `RMU` fuzzing heuristic removes random ratings from the user with the highest influence score and is defined as:

$$\texttt{RMU} : I^{\mathcal{U}} \times D \times \mathbb{Z} \rightarrow 2^{R} \tag{4.4}$$

Items are selected from the set of items rated by that user and both items and rating values are selected uniformly at random. The intuition behind this heuristic is that by removing ratings from the highest influence user, we can cause the user to become non-influential, which may cause the recommendations of other users to change.

`CMU`. The `CMU` fuzzing heuristic changes random ratings by the user with the highest influence score to have the lowest possible rating value. This heuristic is defined as:

$$\texttt{CMU} : I^{\mathcal{U}} \times D \times \mathbb{Z} \rightarrow 2^{C} \tag{4.5}$$

Items are selected uniformly at random from the set of items rated by that user. The intuition behind this heuristic is that by changing ratings of the highest influence user, we may cause a previously recommended item to no longer be recommended.

`ALI`. The `ALI` fuzzing heuristic adds random ratings to the item with the lowest influence score. We define the function for this heuristic as:

$$\texttt{ALI} : I^{\mathcal{I}} \times D \times \mathbb{Z} \rightarrow 2^{A} \tag{4.6}$$

Users for which to add ratings are selected from the set of users that have not yet rated the least influential item. Both users and rating values are selected uniformly at random. The intuition behind this heuristic is that by adding ratings to the least influential item, we may cause it to become influential, causing recommendations to change.

`AMR`. The `AMR` fuzzing heuristic adds new random ratings to items with average rating values near the most influential rating value. We define the function for this heuristic as:

$$\texttt{AMR} : I^{\mathcal{R}} \times D \times \mathbb{Z} \rightarrow 2^{A} \tag{4.7}$$

Items are selected uniformly at random from the set of items with an average rating within 0.05 of the most influential rating. This selects items very close to the influential rating. We experimented with several values of ϵ, and we chose 0.05 as the value that generally produced the most instability. Users are selected uniformly at random from the set of users that have not rated the selected item. The rating value to add is selected uniformly at random. The intuition behind this heuristic is that by adding random ratings to items with an influential average rating, we can move the average away from the influential rating value to reduce the likelihood that the item will be recommended.

`AMA`. The `AMA` fuzzing heuristic adds low valued ratings to items with the highest aggregate attribute influence and is defined as:

$$\texttt{AMA} : I^{\mathcal{A}_{\mathcal{I}}} \times D \times \mathbb{Z} \rightarrow 2^{A} \tag{4.8}$$

Because items can have multiple attributes, this heuristic aggregates the influence scores of all attributes of an item by summing all of their influence scores. We use a slightly modified attribute influence function in order to negatively weight low influence attributes:

$$I_2^{\mathcal{A}_{\mathcal{I}}}(a) = 2 * \#\{a' \mid I^{\mathcal{A}_{\mathcal{I}}}(a) \geq I^{\mathcal{A}_{\mathcal{I}}}(a')\} - \#\mathcal{A}_{\mathcal{I}} \tag{4.9}$$

The item with the highest aggregate attribute influence is selected for modification. Users are selected uniformly at random from the set of users who have not yet rated the selected item. The intuition behind this modification fuzzing heuristic is that by adding low ratings to items with many highly influential attributes, we can decrease the influence of those attributes and cause items with those attributes to not be recommended.

4.1.3 Differential Stability Oracles

To test the stability of recommender systems, we must define appropriate oracles. We define an oracle as a Boolean predicate:

$$f\{d(rank(u), rank'(u)) \mid u \in \mathcal{U}\} < \delta \tag{4.10}$$

This predicate ensures that a function f applied to the set of distances between users' rankings using the original and modified datasets is below a specified threshold δ. In this work, we consider f to be a function that computes the average distance. We can compute the distance between rankings using a variety of metrics, depending on which types of change we wish to be sensitive to.

We present three possible metrics that may be used for computing the distance between rankings that cover a range of possibly important aspects of change, including the order of the Top-K rankings, the inclusion of items in the Top-K recommendations, and the exclusion of important items in a user's Top-K.

Many metrics can be used to compute the amount of change between recommendations. For this work, we quantify the amount of change between recommendations from modified and unmodified datasets using the following three metrics: *AOD*, *Jaccard*, and *TopOut*. The distance is measured between each user's top-k recommendations when using the original dataset and that user's top-k recommendations when using the modified dataset. The reported values are the average values across all users.

AOD is the average overlap distance, and is defined as:

$$\begin{aligned} &AOD(rank_u, rank'_u) = \\ &\quad 1 - \frac{1}{k}\sum_{d=1}^{k} \frac{\#(\{i \mid rank_u(i) \le d\} \cap \{i \mid rank'_u(i) \le d\})}{d} \end{aligned} \tag{4.11}$$

For the *AOD* metric $rank_u$ is the projection of the *rank* function onto user u when using the original dataset. $rank'_u$ is the projection of the *rank* function onto user u when using the modified dataset. This metric is sensitive to both the items in the rankings, as well as their position. It also weights items lower in the rankings less than items at the top of the list, so swapping items in the top half of the ranking causes a larger change than a new item appearing at rank k.

Jaccard is the jaccard distance, and is defined as:

$$Jaccard(R_u, R'_u) = 1 - \frac{\#(R_u \cup R'_u) - \#(R_u \cap R'_u)}{\#(R_u \cup R'_u)} \tag{4.12}$$

For this metric, R_u is the set of items recommended to user u using the recommender trained on the original dataset and R'_u is the set of items recommended to u after modifications are made to the dataset. This metric is sensitive to changing items, capturing the difference in the set of items that are recommended.

We also introduce the *TopOut* measure. This measure checks whether the top item in the unmodified ranked list has dropped out of the top-k rankings when using the modified dataset. We assume that the top ranked item is likely to be the most difficult to change. Therefore, if this item is not in the ranked list after modifications are added to the dataset, then the recommendations should be considered to have significantly changed. We define this measure as:

$$TopOut(R_u, R'_u) = \begin{cases} 0 & r_1 \in R'_u \\ 1 & otherwise \end{cases} \tag{4.13}$$

R_u is the set of items recommended to user u using the recommender trained on the original dataset and R'_u is the set of items recommended to user u after modifications are made to the dataset. Item r_1 is the top-ranked item ($rank(r_1, u) = 1$) in the ranked list of u. This metric is sensitive only to the top ranked item for a user, which generally has the highest likelihood of being preferred by the user. To change the value of this metric, the top item in the original ranking must not be included in the new top-k ranked list.

4.1.4 Practical Considerations

For this approach to be applicable, certain preconditions must be met. First, the developer must have read and write access to the full dataset that was used to train the recommender system under test. This is a reasonable assumption, as testing will generally be performed by a developer or a dedicated tester of the system, and will thus have access to this data. Second, we assume that additions, removals, or changes are realistic modifications that can occur to a dataset which is the case for most recommendation systems that evolve over time. Third, in defining differential oracles for recommender stability, we assume that identifying a threshold of acceptable instability δ is possible either by using standard or historical measures.

When those preconditions are met, the approach can provide not only better stability estimates than existing approaches but also concrete dataset changes that may cause significant instability. Historical stability estimates can then be used by developers to assess the evolution of their recommender from a robustness perspective and to pinpoint departures from established trends. Developers can also use the concrete dataset changes to determine how best to adjust the existing algorithm underlying the recommendation system to make it more robust to variations in the dataset. Last, the stability estimates and the dataset changes can guide data cleansing procedures (for example, by increasing or decreasing the impact of certain records or aspects) and assist in the definition of data retraining policies after deployment.

4.2 Study

We carried out a study to explore the cost-effectiveness of our fuzzing heuristics, and we also explore how the type of influence and type of modifications used by a fuzzing heuristic affect its ability to find instability-inducing modifications. More specifically, we attempt to answer the following questions:

RQ1: How effective are the different influence models in guiding the generation of instability-inducing modifications?

RQ2: How can we fuzz a recommender system if the algorithm is a black box and the type of underlying influence is unknown?

4.2.1 Study Design

We evaluated our approach to fuzzing recommender systems by applying the selected influence-guided fuzzing heuristics defined in Section 4.1.2 to several recommender systems using a variety of recommender algorithms and a movie ratings dataset. We discuss each of these choices in more detail below.

We evaluated the fuzzing heuristics for three sizes of modification set: 1, 10, and 100. These sizes correspond to changes of 0.001, 0.01, and 0.1 percent of the dataset respectively and were chosen to be much smaller than the size of the dataset. With fewer than 0.1% of the ratings being modified, we would expect the recommendations to exhibit proportionally small amounts of change. For each heuristic, size, and recommender system (to be described next), we generated 100 modification sets. We then trained each recommendation algorithm on the modified dataset and generated Top-10 recommendations for every user. We report the average value of the *TopOut* metric over the 100 generated modifications sets in Tables 4.6, 4.7, and 4.8.

4.2.1.1 Recommender Algorithms

For this study we selected four different recommendation algorithms: User-User, Item-Item, FunkSVD, and LightFM. The algorithms were chosen to represent a variety of recommender techniques ranging from memory-based to model-based, and from content-based to collaborative filtering. We also required that they work with the MovieLens dataset. We describe each of these algorithms in detail in Section 3.4.1.1.

4.2.1.2 MovieLens 100k Dataset

We use the dataset released in 1998 from the MovieLens recommendation system. The dataset is available as a group of tab separated files, containing 100 thousand integer ratings (from one to five) collected from 943 users over a period of eight months on 1682 movies. Each user has rated at least 20 items. More details about the data collection process and the dataset itself are available at `https://grouplens.org/datasets/movielens/100k/` [26].

4.2.1.3 Treatments

To study the effectiveness of influence-guided fuzzing, we compare the defined heuristics against five baseline heuristics: `ZERO`, `ARAND`, `RRAND`, `CBRAND`, and `CTRAND`. These baseline heuristics are shown in Table 4.4. The heuristic `ZERO` performs no modifications to the dataset and is equivalent to retraining the recommender algorithm on the original dataset. The other four heuristics perform random changes to the dataset in which selections of user, item, or rating value are made uniformly at random. These baselines were chosen to control for the effect of the influence model on the user or item choice. By comparing the influence-based heuristics to these random baselines, we can evaluate the effectiveness of using influence models to guide fuzzing of recommender systems.

4.2.1.4 Differential Oracle

Our evaluation reports the results from using the *TopOut* distance metric. Our oracle computes the average distance over all users ($f = avg$). We choose to use *TopOut* for our evaluation because it takes the most energy to change. For example, the FunkSVD recommender has some small instability when re-trained on the same dataset, without modifications. Because *TopOut* is sensitive only to major change, it reports an average distance of 0.00000. However, both *AOD* and *Jaccard* report higher values of instability, at 0.004230 and 0.000137 respectively.

Table 4.4: The Modification Heuristics Used In Our Evaluation.

	Short Name	Description
Baselines	`ZERO`	Perform no modifications.
	`ARAND`	**Add** ratings to random users and items.
	`RRAND`	**Remove** random ratings.
	`CBRAND`	**Change** random ratings to the low value.
	`CTRAND`	**Change** random ratings to the top value.
Influence Guided Heuristics	`AMU`	**Add** a rating with random value to a random item for the most influential user.
	`RMU`	**Remove** a rating from a random item for the most influential user.
	`CMU`	**Change** the rating of a random item to the low value for the most influential user.
	`ALI`	**Add** a rating with a random value to the least influential item.
	`AMR`	**Add** a random rating value to an item with an average rating near the most influential average rating.
	`AMA`	**Add** a rating with the low value to a random user for an item with the most attribute influence.

In this study, we chose instability thresholds that were proportional to the size of the modification set being tested. We used thresholds of $\delta = 0.001$, $\delta = 0.01$, and $\delta = 0.1$ for sets of 1, 10, and 100 modifications, respectively. We chose these values to be 100 times the ratio of the sizes of the modification set and dataset. For instance, 100 modifications is 0.1% of the dataset, so we assert that no more than 10% of users should have their top ranked item fall out of their new recommendations.

4.2.1.5 Methodology

To evaluate efficiency, we measure how long it takes each heuristic to generate a modification. We generate 100 modification sets of size 1, 10, and 100 for each heuristic and report the average time required to generate a modification set of each size.

To compare the effectiveness of influence-guided fuzzing heuristics to random fuzzing, we compute the average *TopOut* instability across 100 generated modification sets for each configuration of recommender and modification set size. We explored modification sets of size 1, 10, and 100, which are small compared to the original dataset. Changing 100 ratings would change only 0.1% of the dataset used in this study. We arbitrarily chose to stop with a maximum modification set size of 100.

4.2.1.6 Threats to Validity

The recommendations produced by recommender systems are dependent on both the recommender algorithm as well as the dataset used to train the system. Because we evaluate our approach with only a single dataset, our results may not generalize to other datasets. That said, this dataset is commonly used and includes user and item attributes for content-based recommendation.

In this work we look at only a small subset of the possible influence functions and modification fuzzing heuristics. There are many other heuristics that can generate instability-inducing modifications or better approximation functions for computing influence. We do show that heuristics can be used to generate instability-inducing modifications more effectively than random methods.

There are many possible metrics for evaluating the distance between recommendations. In this work, we show only the results of one metric, *TopOut*. The results when using *Jaccard* and *AOD* were similar, so we do not report them here.

4.2.2 RQ1: Effectiveness of Influence-Guided Fuzzing

The average *TopOut* instability for each configuration is shown in Tables 4.6, 4.7, and 4.8. We perform a pairwise comparison of the instability induced by each influence-guided fuzzing heuristic to the instability induced by the corresponding random approach

Table 4.5: Time Required to Generate Modification Set.

	Time (seconds) to Make M Modifications					
	1		10		100	
Heuristic	Mean	Variance	Mean	Variance	Mean	Variance
`ARAND`	0.539	0.010	0.608	0.003	2.020	0.097
`AMU`	0.600	0.119	0.673	0.059	1.692	0.067
`ALI`	0.607	0.158	0.801	0.191	1.809	0.210
`AMR`	0.458	0.009	0.580	0.015	1.582	0.024
`AMA`	3.126	0.501	3.053	0.135	3.753	0.134

Table 4.6: Mean *TopOut* Instability for Each Recommender System Using Sets of 1, 10, and 100 Add Modifications. Bolded Values Outperform the Random Baseline. Italicized Values are Worse than the Baseline. Values Marked by an Asterisk Exceed the Instability Threshold.

Configuration Recommender,M	`ZERO`	`ARAND`	`AMU`	`ALI`	`AMR`	`AMA`
User-User,1	*0.000000*	0.000064	*0.003139	*0.003796	***0.008208**	0.000064
User-User,10	*0.000000*	0.002969	0.007063	***0.036554**	***0.066925**	*0.000615*
User-User,100	*0.000000*	0.031294	**0.050244**	***0.309364**	***0.438537**	*0.013966*
Item-Item,1	0.000000	0.000021	0.000021	0.000000	0.000106	0.000032
Item-Item,10	*0.000000*	0.002375	0.001803	*0.000042*	0.000870	*0.000085*
Item-Item,100	*0.000000*	0.026681	0.038929	*0.000583*	*0.006363*	*0.000456*
FunkSVD,1	0.000000	*0.006840	0.000000	*0.001389	***0.035323**	0.000000
FunkSVD,10	0.000000	0.000011	*0.014369	***0.032131**	***0.441304**	0.000021
FunkSVD,100	*0.000000*	0.053446	0.040668	***0.273213**	***0.934740**	*0.000000*
LightFM,1	* *0.007031*	*0.011771	*0.011304	*0.011198	*0.011166	*0.012428
LightFM,10	*0.007031*	*0.011474	*0.012015	*0.011220	*0.011601	***0.014836**
LightFM,100	*0.007031*	0.010933	0.010042	0.010923	0.010233	**0.035355**

using Welch's t-test. Values in these tables are **bold** if their expected instability is greater (with $p < 0.05$) than the random baseline of the same type and are *italicized* if the expected instability is lower than the random baseline. Values marked with an asterisk (*) exceed the instability threshold for that size of modification set.

4.2.2.1 Efficiency of Heuristics

In exploring the overall effectiveness of using inferred influence models as heuristics for generating instability-inducing modifications, we evaluate the efficiency of our in-

Table 4.7: Mean *TopOut* Instability for Remove Modification Fuzzing Heuristics.

Configuration Recommender,M	`RRAND`	`RMU`
User-User,1	0.000032	0.000021
User-User,10	0.000392	0.000509
User-User,100	0.003796	**0.008144**
Item-Item,1	0.000011	0.000011
Item-Item,10	0.000000	**0.000074**
Item-Item,100	0.001601	**0.010138**
FunkSVD,1	0.000000	0.000000
FunkSVD,10	0.000000	0.000000
FunkSVD,100	0.000000	**0.062704**
LightFM,1	*0.009544	*0.009873
LightFM,10	*0.010806	*0.010742
LightFM,100	0.010753	0.011389

Table 4.8: Mean *TopOut* Instability for Change Modification Fuzzing Heuristics.

Configuration Recommender,M	`CBRAND`	`CTRAND`	`CMU`
User-User,1	0.000064	0.000032	0.000095
User-User,10	0.001082	*0.000520*	0.000774
User-User,100	0.011113	*0.007041*	*0.006914*
Item-Item,1	0.000053	0.000000	0.000191
Item-Item,10	0.000297	*0.000032*	0.004242
Item-Item,100	0.004952	*0.000329*	**0.066341**
FunkSVD,1	0.000000	0.000000	0.000000
FunkSVD,10	0.000000	0.000000	*0.020901
FunkSVD,100	0.000000	0.000000	***0.243849**
LightFM,1	*0.006946	*0.006840	*0.007285
LightFM,10	0.007116	0.007126	0.007010
LightFM,100	0.006925	0.006978	0.007497

fluence based heuristics based on how long it takes each heuristic to generate a set of modifications. We report results for the heuristics that generate Add modifications. The cost of fuzzing heuristics for Change and Remove modifications were similar. The mean and variance over the course of 100 trials for each heuristic are reported in Table 4.5. We see that the time to generate modification sets for random and influence based fuzzing heuristics are within a factor of six in the worst case, but are practically the same on average considering that the time it takes to train the recommender system on the new data is generally much greater. For example, depending on the recommender algorithm, training on the MovieLens 100k dataset and producing recommendations for all users takes two minutes on average. We see that `AMA` takes considerably longer than other heuristics. This is due to the overhead incurred by aggregating the influence of multiple attributes for every item (see Equation 4.9).

In terms of modification set sizes, *influence-guided fuzzing is efficient at generating small sets of modifications that induce higher amounts of instability in the recommendations.* For instance, generating sets of 100 modifications with `ARAND` for the User-User recommender averages a *TopOut* instability of 0.031. So, we can expect that 3.1% of users will have their top ranked item fall out of their Top-10 list and not be recommended after 100 random ratings are added to the dataset. In contrast, the `ALI` heuristic averages a *TopOut* value of 0.037 with only 10 modifications for the same recommender system. Similarly, for the LightFM recommender, the expected value of *TopOut* for `ARAND` with 100 modifications is 0.011, while the `AMA` heuristic achieves a higher *TopOut* value with a modification set of size 10.

4.2.2.2 Effectiveness of Heuristics

Overall, using influence-guided fuzzing is more effective at generating modification sets that cause instability than randomly generating modifications, causing distances *between recommendations up to 20 times greater than random fuzzing.* We see that, for every recommender system, at least one modification heuristic performs significantly better than the random baseline for modification sets with a size of at least 10. For two recommenders, User-User and FunkSVD, the `AMR` heuristic performs significantly better than the random baseline for all modification sets with at least one modification.

Influence-guided fuzzing is also more effective at detecting unstable recommender systems than random fuzzing. Using the instability thresholds defined in Section 4.2.1.5, we see that, for User-User and FunkSVD, many influence-guided fuzzing heuristics were able to induce instability even when random fuzzing could not. For the User-User recommender, we also see that random fuzzing never exceeds the acceptable threshold. However, two influence-guided fuzzing heuristics, `ALI` and `AMR`, generate sets of modifications that exceed the acceptable level of change for all three sizes of modification set, and a third (`AMU`) is able to show instability for modification sets with a single modification. For all of the recommender systems studied here, if random fuzzing detected instability, then at least one influence-guided fuzzing heuristic also detected the instability.

4.2.2.3 Effectiveness by Influence Model

The effectiveness of a given influence-guided fuzzing heuristic depends on how closely the inferred influence model used by the heuristic approximates the actual influences used by the recommender algorithm. For instance, in Table 4.6 we see that the `AMA` heuristic is not effective at generating instability-inducing inputs for the User-User, Item-Item, and FunkSVD recommenders. This is likely because `AMA` uses an inferred attribute influence model, while the three algorithms for which it does not perform well do not rely on any item attribute data. However, for a recommender that does rely on item attribute information, such as LightFM, the `AMA` heuristic is effective at generating instability-inducing modifications.

The rating influence based `AMR` heuristic is unusually effective at causing significant changes to recommendations for the User-User and FunkSVD recommenders. We believe this is because influential rating values for these recommenders are on the extreme ends of the rating scale. The most influential rating value for the User-User recommender is 1.0, while the influential rating value for the FunkSVD recommender is 5.0. For the other two recommenders, for which `AMR` performed poorly, the most influential rating value was closer to the middle of the scale. The Item-Item recommender had a most influential rating value of 4.0, while the LightFM recommender had an influential rating value of 2.0.

4.2.2.4 Effectiveness by Modification Type

For three of the recommender systems explored in this work, *Addition-type modifications were the most effective at influencing change.* For each of these recommenders, at least one heuristic performed significantly better than random for modification sets of size 10 and 100. For example, the `AMA` heuristic is effective at fuzzing instability-inducing modification sets for the LightFM recommender. For both the User-User and FunkSVD algorithm, the `AMR` heuristic is also effective at generating single modifications that induce significant change in the recommendations. However, Addition-type modifications were not effective at inducing instability for the Item-Item recommender.

Influence-guided heuristics that performed remove modifications were more effective than random for sets of 100 modifications for User-User, Item-Item, and FunkSVD. This modification type was also effective for the Item-Item recommender when fuzzing sets of 10 modifications.

Overall, influence-guided heuristics that generate rating change modifications affected the fewest recommender systems, but when they were effective, they caused large amounts of change. Change modifications were effective only on two algorithms, Item-Item and FunkSVD, and only for sets of 100 modifications, however, they were able to produce higher levels of instability in the Item-Item recommender than any other type of modification. We conjecture that this is because the value of ratings in these systems is more influential than the relationships between users or items.

4.2.3 RQ2: How Effective is Fuzzing in the Absence of Algorithm Influence Information?

We explore whether influence-guided fuzzing is an effective technique for testing stability when the recommender algorithm under test is a black box, and the sorts of influence used by the system are unknown. We introduce using a portfolio approach to generating modifications. We assume a budget of n modifications and a user specified portfolio of m influence-guided fuzzing heuristics. For each modification in the modification set, we use a round robin approach to select one of the m fuzzing heuristics to generate the modification.

We evaluate the effectiveness of this approach with a portfolio of the four influence-guided fuzzing heuristics that generate Addition modifications defined in Section 4.1.2. We call this hybrid heuristic `APORTFOLIO`. Using budgets of $n = 10$ and $n = 100$, we compare the effectiveness of `APORTFOLIO` to `ARAND` using each of the four recommenders studied above. We do not use $n = 1$ for this approach because it is equivalent to selecting a single heuristic, whose values are reported in Table 4.6. We also compare `APORTFOLIO` to the heuristic that produces the most instability for each configuration. The average *TopOut* instability is presented in Table 4.9.

Using a portfolio of influence-guided fuzzing heuristics is effective at generating

Table 4.9: Mean *TopOut* Instability for `APORTFOLIO`

Configuration Recommender,M	`ARAND`	Best Heuristic	`APORTFOLIO`
User-User,10	0.002969	***0.066925**	***0.035599**
User-User,100	0.031294	***0.438537**	***0.257794**
Item-Item,10	0.002375	0.001803	0.000244
Item-Item,100	0.026681	0.038929	*0.008653*
FunkSVD,10	0.000011	***0.441304**	***0.130721**
FunkSVD,100	0.053446	***0.934740**	***0.768982**
LightFM,10	*0.011474	***0.014836**	*0.012450
LightFM,100	0.010933	**0.035355**	**0.017678**

modification sets that cause significant changes in recommendations. As seen in Table 4.9, for two of the recommenders (User-User and FunkSVD), the portfolio approach is able to find modification sets that cause more instability than `ARAND` for both sizes of modification set tested. Additionally, `APORTFOLIO` is able to find sets of modifications that cause more instability than `ARAND` for the LightFM recommender when the budget is 100 modifications. `APORTFOLIO` was not effective for the Item-Item recommender system. This is likely because none of the Add heuristics performed well on the Item-Item recommender.

Chapter 5

Conclusions and Future Work

In this thesis we presented a general approach for the characterization of recommender systems (Chapter 3) which offers a richer view of system behavior than precision or recall. We showed how simple properties defined using our approach relate to properties identified in prior work as being useful for evaluating recommender behavior. We evaluated our approach on five recommender algorithms applied to the MovieLens and Jester datasets and found that our property instantiations can offer insights into the differing behaviors of recommender systems, beyond those of precision and recall. We also show that, when compared across an evolving dataset, our properties offer insights into the robustness of behaviors of a given recommender system as the underlying data evolves.

The property templates and instantiations that we presented in this work represent only a sample of the defined space. In future work, we will perform a more exhaustive exploration of this space as it may reveal additional useful properties for recommender systems, as well as an investigation of the effect of different dataset distributions on the properties. Longer term, we want to use the properties to assist in the explanation of certain recommendations that do not meet a developer's expectations.

In Chapter 4, we presented an approach that uses influence-guided fuzzing to test the stability of recommender systems. We build on the insight that influence models can be inferred from the recommendations produced by a recommender system and the dataset used to train that system. We define fuzzing heuristics that use these inferred influence models to generate modifications to the original dataset that induce instability in the recommendations. To test instability we define a test oracle based on a threshold of acceptable instability, based on measures of distance between users' recommendations. Our study shows that influence-guided fuzzing is effective at finding small sets of modifications that cause significantly more instability than random approaches.

The influence models, heuristics, and instability metrics presented in this work are only a sample of those possible, and were chosen to illustrate the general effectiveness of influence-guided fuzzing of recommender systems. In future work, we will perform a more exhaustive search of this space to identify what features of heuristics and influence models are most effective. For example, we will explore hybrid forms of influence between multiple types of aspects in the dataset, and explore fuzzing heuristics that take advantage of multiple types of influence.

Longer term, we want to use the inferred influence models to explain anomalous behavior of recommender systems, such as the accuracy of a recommender system dropping significantly after retraining or a previously popular item not being recommended to any user. We conjecture that inferred influence models can help a developer understand the decisions underlying a recommenders' behavior.

Bibliography

[1] Iso/iec/ieee international standard - systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010. 2.4

[2] G. Adomavicius and Y. Kwon. Improving aggregate recommendation diversity using ranking-based techniques. *IEEE Transactions on Knowledge and Data Engineering*, 24(5):896–911, May 2012. 2.2

[3] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, June 2005. 2.1

[4] Gediminas Adomavicius and Alexander Tuzhilin. *Context-Aware Recommender Systems*, pages 191–226. Springer US, Boston, MA, 2015. 2.1

[5] Gediminas Adomavicius and Jingjing Zhang. On the stability of recommendation algorithms. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 47–54, New York, NY, USA, 2010. ACM. 1, 2.3

[6] Gediminas Adomavicius and Jingjing Zhang. Stability of recommendation algorithms. *ACM Trans. Inf. Syst.*, 30(4):23:1–23:31, November 2012. 2.3, 4

[7] Marko Balabanović and Yoav Shoham. Fab: Content-based, collaborative recommendation. *Commun. ACM*, 40(3):66–72, March 1997. 2.1

[8] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutirrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109 – 132, 2013. 2.1

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM. 2.4

[10] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 43–52, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. 2.1, 3.4.1.1

[11] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, pages 39–46, New York, NY, USA, 2010. ACM. 2.2

[12] Mukund Deshpande and George Karypis. Item-based top-n recommendation algorithms. *ACM Trans. Inf. Syst.*, 22(1):143–177, January 2004. 3.4.1.1

[13] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press. 2.4

[14] Michael D. Ekstrand, Michael Ludwig, Joseph A. Konstan, and John T. Riedl. Rethinking the recommender research ecosystem: Reproducibility, openness, and lenskit. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, pages 133–140, New York, NY, USA, 2011. ACM. 3.4.1.1

[15] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association. 2.4

[16] Simon Funk. Netflix update: Try this at home, December 2006. 3.4.1.1

[17] Mouzhi Ge, Carla Delgado-Battenfeld, and Dietmar Jannach. Beyond accuracy: Evaluating recommender systems by coverage and serendipity. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, pages 257–260, New York, NY, USA, 2010. ACM. 2.2

[18] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM. 2.4

[19] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008. 2.4

[20] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, December 1992. 2.1

[21] Kenneth Y. Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Inf. Retr.*, 4(2):133–151, 2001. 3.4.1.2

[22] Nathaniel Good, J. Ben Schafer, Joseph A. Konstan, Al Borchers, Badrul Sarwar, Jon Herlocker, and John Riedl. Combining collaborative filtering with personal agents for better recommendations. In *Proceedings of the National Conference on Artificial Intelligence and the Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, pages 439–446, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence. 1, 2.2

[23] Asela Gunawardana and Guy Shani. *Evaluating Recommender Systems*, pages 265–308. Springer US, Boston, MA, 2015. 2.3

[24] Ihsan Gunes, Cihan Kaleli, Alper Bilge, and Huseyin Polat. Shilling attacks against recommender systems: a comprehensive survey. *Artificial Intelligence Review*, 42(4):767–799, Dec 2014. 2.3

[25] Donna Harman. Is the Cranfield paradigm outdated? In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1–1, New York, NY, USA, 2010. ACM. 2.2, 3

[26] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015. 3.4.1.1, 3.4.1.2, 4.2.1.2

[27] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 230–237, New York, NY, USA, 1999. ACM. 1, 2.1, 2.2

[28] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, January 2004. 1, 2.2, 3

[29] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, October 2002. 2.2

[30] P. Koopman. Toward a scalable method for quantifying aspects of fault tolerance, software assurance, and computer security. In *Proceedings Computer Security, Dependability, and Assurance: From Needs to Solutions (Cat. No.98EX358)*, pages 103–131, 1998. 2.4

[31] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, Aug 2009. 2.1

[32] Maciej Kula. Metadata embeddings for user and item cold-start recommendations. In Toine Bogers and Marijn Koolen, editors, *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.*, volume 1448, pages 14–21. CEUR-WS.org, 2015. 2.1, 3.4.1.1

[33] Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. Principles of explanatory debugging to personalize interactive machine learning. In *Proceedings of the 20th International Conference on Intelligent User Interfaces*, pages 126–137. ACM, 2015. 3

[34] Shyong K. Lam and John Riedl. Shilling recommender systems for fun and profit. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 393–402, New York, NY, USA, 2004. ACM. 2.3

[35] Daniel Lemire and Anna Maclachlan. *Slope One Predictors for Online Rating-Based Collaborative Filtering*, pages 471–475. 2005. 3.4.1.1

[36] Benjamin Letham, Cynthia Rudin, Tyler H McCormick, David Madigan, et al. Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model. *The Annals of Applied Statistics*, 9(3):1350–1371, 2015. 3

[37] Xin Luo, Yunni Xia, and Qingsheng Zhu. Incremental collaborative filtering recommender based on regularized matrix factorization. *Knowledge-Based Systems*, 27:271 – 280, 2012. 2.1

[38] Sean M. McNee, John Riedl, and Joseph A. Konstan. Being accurate is not enough: How accuracy metrics have hurt recommender systems. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, pages 1097–1101, New York, NY, USA, 2006. ACM. 2.2

[39] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990. 2.4

[40] Bamshad Mobasher, Robin Burke, Runa Bhaumik, and Chad Williams. Toward trustworthy recommender systems: An analysis of attack models and algorithm robustness. *ACM Trans. Internet Technol.*, 7(4), October 2007. 2.3

[41] Michael O'Mahony, Neil Hurley, Nicholas Kushmerick, and Guénolé Silvestre. Collaborative recommendation: A robustness analysis. *ACM Trans. Internet Technol.*, 4(4):344–377, November 2004. 2.3, 4

[42] Michael J. Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review*, 13(5):393–408, Dec 1999. 2.1

[43] Pearl Pu, Li Chen, and Rong Hu. A user-centric evaluation framework for recommender systems. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, pages 157–164, New York, NY, USA, 2011. ACM. 1, 2.2

[44] Al Mamunur Rashid. *Mining Influence in Recommender Systems*. PhD thesis, Minneapolis, MN, USA, 2007. AAI3250160. 4.1.1

[45] Al Mamunur Rashid, George Karypis, and John Riedl. *Influence in Ratings-Based Recommender Systems: An Algorithm-Independent Approach*, pages 556–560. 4.1.1

[46] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, pages 175–186, New York, NY, USA, 1994. ACM. 3.4.1.1

[47] Paul Resnick and Hal R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, March 1997. 1

[48] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2Nd ACM Conference on Electronic Commerce*, pages 158–167, New York, NY, USA, 2000. ACM. 2.2, 3

[49] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Application of dimensionality reduction in recommender system-a case study. Technical report, Minnesota Univ Minneapolis Dept of Computer Science, 2000. 2.1

[50] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the Inter-*

national Conference on World Wide Web, pages 285–295, New York, NY, USA, 2001. ACM. 2.1, 3.4.1.1

[51] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. *Collaborative Filtering Recommender Systems*, pages 291–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 2.1

[52] Ali Shahrokni and Robert Feldt. A systematic review of software robustness. *Information and Software Technology*, 55(1):1 – 17, 2013. Special section: Best papers from the 2nd International Symposium on Search Based Software Engineering 2010. 2.4

[53] Guy Shani and Asela Gunawardana. *Evaluating Recommendation Systems*, pages 257–297. Springer US, Boston, MA, 2011. 1, 2.2, 3.2

[54] David Shriver, Sebastian Elbaum, Matt Dwyer, and David Rosenblum. Characteristic properties of recommendation systems, 2018. 1.2

[55] David Shriver, Sebastian Elbaum, Matt Dwyer, and David Rosenblum. Influence-guided fuzzing for testing the stability of recommender systems. 2018. 1.2

[56] Ian Soboroff and Charles Nicholas. Combining content and collaboration in text filtering. In *Proceedings of the IJCAI*, volume 99, pages 86–91. Citeseer, 1999. 2.1

[57] Saúl Vargas and Pablo Castells. Rank and relevance in novelty and diversity metrics for recommender systems. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, pages 109–116, New York, NY, USA, 2011. ACM. 1, 2.2

[58] W. Woerndl and G. Groh. Utilizing physical and social context to improve recommender systems. In *2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Workshops*, pages 123–128, Nov 2007. 2.1

[59] Alvin W. Wolfe. Social network analysis: Methods and applications. *American Ethnologist*, 24(1):219–220, 1997. 4.1.1

[60] Y. Y. Yao. Measuring retrieval effectiveness based on user preference of documents. *J. Am. Soc. Inf. Sci.*, 46(2):133–145, March 1995. 2.2

[61] Tao Zhou, Zoltn Kuscsik, Jian-Guo Liu, Mat Medo, Joseph Rushton Wakeling, and Yi-Cheng Zhang. Solving the apparent diversity-accuracy dilemma of recommender systems. *Proceedings of the National Academy of Sciences*, 107(10):4511–4515, 2010. 1, 2.2

[62] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. Improving recommendation lists through topic diversification. In *Proceedings of the 14th International Conference on World Wide Web*, pages 22–32, New York, NY, USA, 2005. ACM. 1, 2.2