Towards Recommender Engineering

Tools and Experiments for Identifying Recommender Differences

A Dissertation

SUBMITTED TO THE FACULTY OF THE

UNIVERSITY OF MINNESOTA

BY

Michael D. Ekstrand

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Joseph A. Konstan

July, 2014

Typeset with XeLaTeX. Body text set in TeX Gyre Termes, section headings in TeX Gyre Heros[1], sans-serif contrast in Gillius, and code in Ubuntu Mono.

[1] TeX Gyre Termes and Heros are free OpenType equivalents to Times and Helvetica, respectively.

Dedicated in memory of

John T. Riedl, Ph.D

professor, advisor, mentor, friend

Acknowledgements

Many people have helped me along the road to completing my Ph.D. Joe Konstan and John Riedl have been amazing advisors, who have helped me develop as both a researcher and a person. I have also greatly benefited from the feedback and teaching of the rest of my committee: John Carlis, Arindam Banerjee, and John Logie.

Thank you to my colleagues in GroupLens for helping me refine my ideas and generally creating an awesome environment in which to work and study. And the conversations at the lunch table (and elsewhere) with Aaron Halfaker, Morten WArcke-Wang, Daniel Kluver, Kenny Shores, Jacob Thebault-Speaker, and whoever else joined us on a particular day day have greatly helped my research and life. Thank you as well to the GroupLens alumni: your legacy carries on in the quality of idea generation and refinement, you looking out well for us younger researchers. It is great to be a part of such a family. I would like to particularly thank Sean McNee, who told me to unit test my research code, and Jon Herlocker, who reminded me not to spend too much time profiling dot products; more importantly, their research and scholarship on recommender systems has been profoundly influential on my own. Rich Davies's input was invaluable in working through the early design of LensKit.

This work is the result of collaborations with many excellent researchers. Martijn Willemsen taught me how to robustly measure what people think, and has patiently helped us computer scientists learn to do psychology. Jack Kolb has been very helpful in developing LensKit and conducting experiments with it, and Lingfei He has carried on that work well. It has been very good to work with Michael Ludwig on engineering LensKit and designing and building Grapht. Max Harper's support and feedback in making LensKit usable and running the live MovieLens experiment was indispensable.

I would also not be successfully turning in and defending this dissertation without the support of my wife, Jennifer, who has provided much encouragement and helped me refine my ideas and their presentation. My parents also, for teaching me to read and think. And our friends at church for supporting Jennifer and I through a difficult semester.

Finally, I would like to thank JetBrains for making an unbelievably excellent Java development environment, and YourKit for their excellent profiler (and providing a free license for LensKit development), and Sweet Maria's for selling me such excellent coffee. This work was also supported by the National Science Foundation under a variety of grants: IIS 05-34939, 08-08692, 08-12148, and 10-17697.

Abstract

Since the introduction of their modern form 20 years ago, recommender systems have proven a valuable tool for help users manage information overload. Two decades of research have produced many algorithms for computing recommendations, mechanisms for evaluating their effectiveness, and user interfaces and experiences to embody them. It has also been found that the outputs of different recommendation algorithms differ in user-perceptible ways that affect their suitability to different tasks and information needs [Mcn+02]. However, there has been little work to systematically map out the space of algorithms and the characteristics they exhibit that makes them more or less effective in different applications. As a result, developers of recommender systems must experiment, conducting basic science on each application and its users to determine the approach(es) that will meet their needs.

This thesis presents our work towards *recommender engineering*: the design of recommender systems from well-understood principles of user needs, domain properties, and algorithm behaviors. This will reduce the experimentation required for each new recommender application, allowing developers to design recommender systems that are likely to be effective for their particular application.

To that end, we make four contributions: the LensKit toolkit for conducting experiments on a wide variety of recommender algorithms and data sets under different experimental conditions (offline experiments with diverse metrics, online user studies, and the ability to grow to support additional methodologies), along with new developments in object-oriented software configuration to support this toolkit; experiments on the configuration options of widely-used algorithms to provide guidance on tuning and configuring them; an offline experiment on the differences in the errors made by different algorithms; and a user study on the user-perceptible differences between lists of movie recommendations produced by three common recommender algorithms. Much research is needed to fully realize the vision of recommender engineering in the coming years; it is our hope that LensKit will prove a valuable foundation for much of this work, and our experiments represent a small piece of the kinds of studies that must be carried out, replicated, and validated to enable recommender systems to be engineered.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Recommender engineering is the practice of directly designing and implementing recommender systems to meet particular user information needs or business objectives from well-understood principles. These principles encompass the requirements surrounding various user information needs; the properties of recommendation domains that affect the utility of recommendation; the characteristics of different algorithms for doing recommendation, filtering, and information retrieval; and the way in which these properties interact to affect the suitability of a complete recommendation solution to its intended purpose.

When this practice is feasible, the recommender engineer will build a new recommender system through a standard engineering process:

1. Assess the requirements of various stakeholders in the recommender system, particularly the users and the business or organization operating the system.

2. Analyze the recommendation domain — movies, music, job listings, books, research papers, a bit of everything — to identify properties that will affect the suitability of various approaches. The *Handbook of Recommender Engineering* will provide guidance on what properties to look for, such as sparsity (how many items are consumed by the typical user), consumption cost, homogeneity, and perhaps many more.

3. Select and configure algorithms — or a combination of algorithms — that will meet

the stakeholder requirements in the given domain when embedded in a suitable user experience.

The recommender systems community is, for better or worse, a long way from making this vision a reality. Recommender systems provide users with useful recommendations in many domains, and many techniques provide measurable benefit across various applications, but there is not yet a systematic understanding of why particular recommender techniques work here or there, or what makes one technique preferable to another for any given application. We could say that we know *that* recommendation works, but have little understanding as to *how*.

Currently, recommender system developers have essentially two choices. The simplest is to use an off-the-shelf algorithm, such as item-based collaborative filtering, that works in many domains and blindly apply it. This will give them a working recommender system that is probably better than no recommender — and will provide useful results in many applications — but is not particularly well-tuned to any one of them. We have little reason to believe that recommendation is a one-size-fits-all problem, and evidence that different applications do indeed call for different techniques [Mcn+02].

The other, more demanding option is to build a specialized recommender for the particular application (e.g. recommending job listings to users of LinkedIn) and conducting extensive simulations, field trials, and perhaps user studies to determine the best approach. This option effectively requires each system builder to carry out basic scientific research for every new recommender application; there is little understanding available to provide guidance as to what approaches are likely to work better or worse in a given situation, so they must try things and see what works.

Recommender engineering will allow developers to build specialized, high-quality so-

lutions to many recommendation problems without needing to conduct basic research for each application. There will still be need for research on how to improve our techniques and work in more exotic domains, and the high end of recommender development will still likely need significant on-site study, but the overall quality of recommendation in day-to-day use will hopefully improve.

In order to be able to engineer recommenders, however, we need significant advances in knowledge on several fronts, including:

- What types of recommendations are needed to meet different types of user needs? When do users need high novelty or exhaustive coverage of the domain?
- What properties of recommendation domains affect the performance of different techniques towards meeting user needs?
- What are the tendencies and 'behaviors' of different recommendation techniques, and what kinds of needs can they most suitably meet? The behaviors of algorithms may well interact with the properties of the domain in subtle ways.
- How does the user experience — recommendation interface, preference elicitation or inference system, etc. — interact with algorithm behavior, domain properties, and tasks to affect the system's overall suitability?

Previous work has found that different recommender algorithms exhibit identifiably different behaviors [Mcn+02; MKK06], and that these behaviors affect the algorithms' suitability for different recommendation tasks. However, there is much work to be done to systematically map out the space of tasks, domains, and algorithms, and identify the properties that determine a recommender's suitability.

While generalizability is a hallmark of scientific inquiry in general, it is particularly important for the research needed to enable recommender engineering. We cannot study every possible application and task in the research setting. Therefore, we need to study a wide variety of tasks, domains, and algorithms in such a way so as to identify the questions that need to be asked of new tasks and domains in order to design an effective recommender. If, for example, the sparsity and diversity of the item space consistently affect the suitability of particular techniques, then it is reasonable to identify those properties as key; an engineer can then use the sparsity and diversity of a completely new domain as a guide to finding a good recommendation solution.

It is also necessary for recommender systems research to be reproducible, so that the science is of high quality, and the results can be validated and more easily generalized. It is difficult to test the generalizability of a research result to a new domain if the original result cannot easily be replicated at all.

Recommender engineering is unlikely to eliminate the need to test proposed solutions for deployed applications. It should, however, significantly decrease the search space that developers must consider. It is also possible that it will, in the end, prove to be an elusive goal; that the noise of user behavior or other factors make it difficult to reduce recommendation to a manageable set of characteristics. We contend, however, that it is still a worthwhile goal in the medium term. Even if recommender system developers must still conduct extensive experiments in 20 years' time, the research needed to build the kind of understanding of recommendation that might enable engineering will still improve our understanding of the problem greatly, and contribute to our scientific understanding of the way users interact with information systems.

This dissertation is focused on enabling and launching the research that will hopefully lead to recommender engineering becoming a reality. There is, as we have argued, a great

deal of work to do, and we anticipate that it will be at least a decade before it is possible to write a *Handbook of Recommender Engineering*. In this thesis, we make advance the state of the art in two necessary directions:

- Improving the reproducibility of recommender systems research and the usability of research results.
- Examining how recommender algorithms differ in ways that could affect their suitability to different users or tasks.

The rest of this thesis is organized as follows:

- In Chapter 2, we survey related work and background literature on recommender systems and their design evaluation.
- Chapter 3 presents LensKit, an open-source software package we have developed for reproducible recommender systems research. LensKit provides a common platform for developing algorithms, measuring their performance on different data sets, and comparing new ideas against current best practices. These capabilities are crucial to supporting the wide range of research we envision in a reproducible and robustly documented fashion.
- Chapter 4 describes a new dependency injection framework we have developed with novel capabilities for configuring object-oriented programs made up of many composable components.
- Chapter 5 documents experiments on the impact of different configuration options for common recommender algorithms, in the spirit of previous comparative evaluations of recommender system designs [HKR02]. The goal of this work is to provide insight

into how to configure and tune different recommender algorithms, with the particular goal of developing systematic strategies for recommender parameter tuning that can someday be automated. Engineering a recommender is not just a matter of picking the right algorithm family (or families); there are many configuration points and design decisions to be made for any individual algorithm. This chapter — and future work like it — helps reduce the search space the recommender engineer (or future scientist) must consider.

- Chapter 6 explores whether and when different algorithms make different mistakes. This provides further evidence for the existence of interesting differences between algorithms that affect their suitability for different users, items, or applications, and provides some insight into where those differences might be.

- Chapter 7 presents a user study we ran to identify differences that users perceive between the recommendations produced by different commonly-used recommender algorithms. This work makes progress on identifying algorithm differences that matter to user needs, and does so in a way that extracts particular relationships between algorithms, properties, and user satisfaction that can be validated in further studies on additional domains and applications. Extracting such relationships is important to enable future research to establish more easily which results are general and which are domain-specific behaviors.

- Finally, chapter 8 summarizes our findings and maps out some future work.

This work advances the state of recommender research and provides a foundation for extensive further research on how to best meet particular user needs with recommendation.

Chapter 2

Background and Vision

Recommender systems [Ric+10; ERK10] are computer-based tools for suggesting items to users. They often have some form of personalization, attempting to find items that the particular user will like; a recommender may operate in a persistent or even mixed-initiative [Hor99] fashion, suggesting items to the user as they interact with the service in which the recommender is embedded, or with more explicit input, such as recommending items to go with a shopping cart or personalizing search results.[1]

Information retrieval and recommender systems are closely related, sharing many algorithmic and evaluation methods [Sar+02; Hof04; Bel12]. Modern search services are also being increasingly personalized, bringing the fields even closer together. Daniel Tunkelang once quipped that '[recommender systems are] just search with a null query'[2].

The idea of using computers to recommend items that particular users may enjoy has been around for many years; very early work by Rich [Ric79] provided book recommendations to library patrons over a remote computer terminal. Their modern form was first introduced in the early 1990s with collaborative filtering; GroupLens [Res+94], Ringo [SM95], and BellCore [Hil+95] mined the preferences of a large group of users to generate recommendations. Recommender systems have had significant commercial impact [SKR01],

[1] Portions of this work have been published in [ERK10] and [Eks14].

[2] @dtunkelang on Twitter: '@alansaid @xamat @mitultiwari I'm not really a #recsys guy – am more of a search guy. Though #recsys is just search with a null query. :-)' (https://twitter.com/dtunkelang/status/389438505270521856)

playing an important role in both late-90's dot-com boom and the current generation of online services. The have also been a subject of continual research interest for two decades, and the subject of a dedicated conference (ACM Recommender Systems) since 2007.

2.1 Recommender Algorithms

One of the foundational approaches to recommendation is *collaborative filtering*: using the preferences of other users to determine what should be recommended to the active user. This takes several forms:

- User-based collaborative filtering [Res+94; Her+99] computes recommendations for a user by finding other users with similar preferences and recommending the things they like.

- Item-based collaborative filtering [Sar+01; LSY03; DK04] derives a notion of item similarity from user rating or purchase behavior and recommends items similar to those the user has already said they like.

- Matrix factorization methods decompose the user × item matrix of preference data into a more compact, denser representation that can be used to extrapolate the expected preference of items the user has not encountered. One of the most common of these techniques is singular value decomposition [Dee+90; Sar+02]; gradient descent has proven to be an effective way of factorizing the matrix in a manner that is computationally efficient and useful for recommendation but does not preserve all the mathematical properties of a proper singular value decomposition [Fun06; Pat07]. Other techniques based on matrix decomposition have included factor analysis [Can02] and eigenvalue decomposition [Gol+01].

- Probabilistic methods interpret the rating data in a probabilistic fashion. Many of these methods are also matrix factorization methods, such as probabilistic latent semantic indexing [Hof04; JZM04], probabilistic matrix factorization [SM08], and latent Dirichlet allocation [BNJ03].

In addition to collaborative filtering, recommender systems have been built on many other ideas as well. Content-based filtering uses the content and metadata of items to provide recommendations [BS97; MR00; SVR09]. This has the advantage of not requiring an extensive set of user behavior in order to do recommendation, but can only capture aspects of user preference that can be explained using the available item information. Recommendation can also be viewed as an information retrieval [Bel12] or machine learning problem; many standard machine learning techniques for classification and ranking have been applied to recommendation, including Bayesian networks [CG99; ZZ06], Markov decision processes [SHB05], and neural networks [SMH07].

Hybrid recommender systems [Bur02] combine two or more different recommender algorithms to create a composite. In some applications, hybrids of various types have been found to outperform individual algorithms [Tor+04]. Hybrids can be particularly beneficial when the algorithms involved cover different use cases or different aspects of the data set. For example, item-item collaborative filtering suffers when no one has rated an item yet, but content-based approaches do not. A hybrid recommender could use description text similarity to match the new item with existing items based on metadata, allowing it to be recommended anyway, and increase the influence of collaborative filtering as users rate the item; similarly, users can be defined by the content of the items they like as well as the items themselves. Fab used such an approach, matching items against both the content of items liked by the user and the content of items liked by similar users [BS97].

Burke [Bur02] provides a thorough analysis of hybrid recommender systems, grouping them into seven classes:

- *Weighted* recommenders take the scores produced by several recommenders and combine them to generate a recommendation list (or prediction) for the user.
- *Switching* recommenders switch between different algorithms and use the algorithm expected to have the best result in a particular context.
- *Mixed* recommenders present the results of several recommenders together. This is similar to weighting, but the results are not necessarily combined into a single list.
- *Feature-combining* recommenders use multiple recommendation data sources as inputs to a single meta-recommender algorithm.
- *Cascading* recommenders chain the output of one algorithm into the input of another.
- *Feature-augmenting* recommenders use the output of one algorithm as one of the input features for another.
- *Meta-level* recommenders train a model using one algorithm and use that model as input to another algorithm.

Hybrid recommenders proved to be quite powerful in the Netflix Prize [BL07]; the winning entry was a hybrid of some 100 separate algorithms. Hybrids can often squeeze extra accuracy out of the available data for recommendation (sometimes at considerable computational cost [Ama12]). They are also useful to adapt a system to the needs of different users or different items.

Hybrid-like characteristics are not only brought about by combining individual algorithms in weighting schemes; there are many algorithms that can be understood as feature-combining seeing active use and research. Some of them integrate feature-based with collaborative filtering data in a single learning model, often a matrix factorization model [SB10; Che+12].

2.2 Evaluating Recommender Systems

With many different approaches to recommendation, and different domains and tasks to which they can be applied, it is necessary to have some means of evaluating how good a recommender system is at doing its job. The goal of an evaluation is to measure the recommender's ability to meet its core objectives: meet users' information needs, increase lifetime customer value for a service's customer relationships, etc. However, it can be costly to try algorithms on real sets of users and measure the effects. Further, measuring some desired effects may be intractable or impossible, resulting in the need for plausible proxies.

2.2.1 Offline Evaluation

Offline algorithmic evaluations [BHK98; Her+04; GS09] have long played a key role in recommender systems research. They are often used on their own — though not without serious limitations — to assess the efficacy of a recommender system. It is also common to use offline analysis to pre-test algorithms in order to understand their behavior prior to user testing as well as to select a small set of candidates for user testing from a larger pool of potential designs. Since user trials can be expensive to conduct, it is useful to have methods for determining what algorithms are expected to perform the best before involving users.

The basic structure for offline evaluation is based on the train-test and cross-validation

techniques common in machine learning. It starts with a data set, typically consisting of a collection of user ratings or histories and possibly containing additional information about users and/or items. The users in this data set are then split into two groups: the training set and the test set. A recommender model is built against the training set. The users in the test set are then considered in turn, and have their ratings or purchases split into two parts, the *query set* and the *target set*. The recommender is given the query set as a user history and asked to recommend items or to predict ratings for the items in the target set; it is then evaluated on how well its recommendations or predictions match with those held out in the query, or on how effective it is at retrieving items in the target set. This whole process is frequently repeated as in k-fold cross-validation by splitting the users into k equal sets and using each set in turn as the test set with the union of all other sets as the training set. The results from each run can then be aggregated to assess the recommender's overall performance, mitigating the effects of test set variation [GS09]. Some experiments use variants of this model, such as simply splitting the rating tuples into k partitions for cross-validation.

The basic offline evaluation model can be refined to take advantage of the temporal aspects of timestamped data sets to provide more realistic offline simulations of user interaction with the service. The simplest of these is to use time rather than random sampling to determine which ratings to hold out from a test user's profile [GS09]; this captures any information latent in the order in which the user provided ratings. Further realism can be obtained by to restrict the training phase as well, so that in predicting a rating or making a recommendation at time t, the recommendation algorithm is only allowed to consider those ratings which happened prior to t [GS09; LHC09; Bur10]. This comes at additional computational expense, as any applicable model must be continually updated or re-trained as the evaluation works its way through the data set, but allows greater insight into how the

algorithm performs over time.

Offline evaluations measure the performance of the recommender with a variety of metrics. **Prediction accuracy metrics** such as mean absolute error (MAE) [SM95; BHK98; Her+99; Pen+00; HKR02] and root mean squared error (RMSE) [Her+04; BL07] measure the accuracy with which the recommender can predict the user's ratings of the test items. This is applicable to domains where users are providing explicit ratings, as opposed to implicit preference data through their purchasing or reading behavior, and attempts to estimate how good the recommender is at modelling and predicting the user's preferences.

Top-*N* metrics measure the quality of top-*N* recommendation lists produced by the recommender. Many recommenders are used to produce lists of suggested items to show to a user, and top-*N* metrics attempt to measure the recommender's suitability for this task. These metrics are often borrowed from machine learning and information retrieval. Some, such as precision/recall [Sal92], F_1 [Rij79; YL99], and ROC curves and A-measures [Swe63], attempt to measure how good the recommender is at distinguishing relevant items from irrelevant ones. Others, such as mean reciprocal rank (MRR), consider how good the recommender is at putting at least one relevant item near the top of the list. Discounted cumulative gain [JK02] and related measures [BHK98] measure how good the recommender is at putting good items at the top and bad items lower, giving lower weight to position further down the list as users are less likely to even consider the 8^{th} recommendation than the 1^{st}.

Top-*N* metrics as typically deployed have a significant limitation. They require the ability to know, for each recommended item, whether it is good or bad. In supervised learning situations, or in certain information retrieval experiments such as classical TREC competitions, this is not a problem: for a given query, every document has an expert judgement of its relevance. If the retrieval engine suggests a 'not relevant' document, that can be counted

against it.

When evaluating a recommender system, however, we do not often have a fully-coded corpus. If a user did not rate an item, or did not purchase it, we do not know whether they would like it or not. It's more likely that they would dislike it [MZ09], but the premise of recommendation is that such dislike is not a given: there are items the user does not know about but would like. If a recommender finds an item A which the user did not purchase in the available data and suggests it first, followed by an item B which the user did purchase, there are two possibilities. The first is that the user would not like A, and therefore the recommendation is bad. Top-N metrics typically assume this, and 'punish' the recommender for such a recommendation (MRR, for example, measures the rank of the first good item; this increases that rank, decreasing MRR from 1 to 0.5). However, it is also possible that the user would, in fact, like A, possibly in lieu of B. The fact that they did not know about it might even make it a much better recommendation; it is possible they would have found B without the recommender. Therefore, such evaluations can very easily punish the recommender for doing its job. Bellogin [Bel12] provides more detail on the problems and potential remedies for such evaluation structures.

2.2.2 User-Based Evaluation

While offline evaluations allow us to easily and cheaply examine the behavior of recommender algorithms, the true test of a recommender comes when it meets its users. To be useful, a recommender needs to satisfy its users' needs and/or accomplish its business objectives (which ideally flow from satisfying the needs of the business's customers). Therefore, it is necessary to test recommenders with real users.

These kinds of tests take two major forms. Field trials examine the behavior of users as they are ordinarily using the recommender. A/B testing is one kind of field trial used heavily

in industrial applications: give two sets of users of a system different recommenders (or some other treatment, such as a different purchasing interface) and compare them on some key metric (such as the rate at which they like songs that have been played in an Internet radio application).

Other experiments are directly visible to their users, and often involve surveys, prototype interfaces, and other more laboratory-style apparatuses. Such user studies are widely used to evaluate the usefulness of particular recommender applications [Mcn+02; Eks+10] and to answer scientific questions about user interaction with recommender systems [Bol+10]. The design and execution of user studies has improved over time; historically, many studies involved relatively simple user questionnaires (a practice that continues today), but recent years have seen increasing development and use of more sophisticated study designs and analysis techniques [Kni+12].

One such technique, *structural equation modeling* [Kli98; Kni+12], is a powerful tool for investigating the perceived factors that influence user satisfaction and choices. It allows us to not only measure what algorithms or items the user ultimately prefers, but also assess how specific aspects of the recommendations (such as novelty and diversity) influence their preferences and behavior. A user may prefer algorithm A over B because it is diverse and therefore more appropriate to meeting their needs, and SEM allows us to quantify and test these kinds of relationships.

Field trials and user studies allow us to get at different aspects of the recommender's performance. Field trials are often more realistic, as the user is interacting with the recommender in the course of their ordinary work. User studies are often more contrived, although user study techniques can be deployed in a non-contrived fashion, such as by asking users of a live system to take a short survey. Field trials measure the recommender by what users do in response to it: do they listen to more music, buy more things, renew their subscription,

etc. These measurements must be carefully designed, to make sure that they measure things that are useful for assessing the long-term usefulness or business value of a recommender, but they do not have the noise of users not really understanding how they think, or saying they would do one thing when they would really do another. They are limited, however, in their ability to get at *why* users are making particular decisions, to explore the psychological processes involved in a user's interaction with the system. Well-designed user studies can get at these more subjective aspects of the user's experience, measuring the user's satisfaction and user-perceptible properties of the recommender. These subjective measurements can also be correlated with objective measurements of recommender behavior and user activity for a more complete view of the human-recommender interaction [Kni+12].

2.3 Differences in Recommenders

Much research on recommender systems has focused primarily on their accuracy [Her+04]. However, researchers have long recognized that accuracy is not the sole property by which a recommender system should be evaluated. McNee, Riedl, and Konstan [MRK06a; MRK06b] argued for a variety of additional considerations related to the user experience of recommender systems. Further, different algorithms may exhibit different user-perceptible behaviors, even if they perform similarly on accuracy metrics, and that these behaviors affect their suitability for different recommendation tasks [Mcn+02; Tor+04; MKK06].

Several non-accuracy factors have been of recurring and increasing interest in recommender systems research. *Diversity* has been widely considered [Zie+05; VC11; Zho+10; WGK14], measuring the diversity of recommendations by various means and attempting to provide users with more diverse sets of items from which to chose. In addition to providing a more satisfactory set of recommendations, it can also make the decision process

itself easier and more pleasant for users [Bol+10]. Diversity has long been recognized as an important factor in the results of information retrieval systems generally; Clarke et al. [Cla+08] present relatively recent work on evaluating IR systems for novelty and diversity, but diversity (at least in terms of the likely relevance) was also considered by Carbonell and Goldstein [CG98] and much earlier by Goffman [Gof64].

There has also been significant work on the *novelty* of recommendations [ZH08; VC11], trying to provide users with recommendations that they are unlikely to have heard of by other means. This has also been cast as *serendipity* [GDJ10]: the 'happy accident', an item the user did not expect but that turned out to be good. Increasing the number of serendipitous encounters is indeed the purpose of recommender systems, particularly in entertainment domains: if the user already knows about, or would discover through natural means, all the movies they would like, they would have no need for a recommender.

Another consideration is the *stability* of a recommender, the degree to which its recommendations change or remain static over time [Bur02] or its degree of self-consistency [AZ12]. A stable recommender system has a certain degree of predictability, and may be more robust to attack [LR04], but may also be boring: if it takes a lot for recommendations to change, repeat users might not be satisfied.

While it is well-established that there are factors beyond accuracy that affect a recommender's suitability for different tasks, and that different algorithms exhibit different characteristics and are therefore more or less suitable in different situations, there has been inadequate systematic exploration of what these differences are, why they matter, and how we can harness them to build better recommendation solutions. Our empirical work therefore focuses on understanding how recommenders differ, attempting to identify differences between recommender algorithms that are interesting and useful for building better, more targeted solutions, and are of scientific interest to understanding how humans interact with

personalized information retrieval and filtering systems.

2.4 Reproducible and Reusable Research

Reproducibility is one of the cornerstones of the scientific enterprise. The ability to repeat experiments and obtain comparable results is critical to ensuring that our results are reliable, generalizable, and predictive. Reproducibility doesn't come for free, however: reproducible research must be carefully and thoroughly documented and appropriate data, materials, software, and equipment must be available to scientists who may wish to reproduce results. Scientists in a wide variety of disciplines have discussed the necessity and challenges of reproducible research, particularly in the face of the increasing reliance on computation in many disciplines [GL04; PDZ06; Lai+07; Don+09; VKV09; Pen11].

Lougee-Heimer [Lou03] argues for open-source distribution of software embodying research results as a tool to promote reproducibility in the field of operations research. We think his arguments are equally applicable in other research domains where computational techniques are an important research output, including recommender systems. Publishing the code used in a research publication for others to read and execute is the surest way to ensure that they can faithfully reproduce the results, either to validate them or as a starting point for further investigation.

Recommender systems research is, in our experience, often hard to reproduce [Eks+11; Eks14]. There are several causes and consequences of this difficulty:

- It is difficult to understand existing recommender algorithms in enough detail to implement them. Many algorithms are published in the research literature, but the level of detail they provide varies greatly. This makes it difficult in many cases to under-

stand exactly what the original authors did, particularly in edge cases, in enough detail to re-implement the algorithm and make it useful.

- It is difficult to reproduce and compare research results. In addition to the variation in their descriptions of algorithms, recommender research papers do not always specify their evaluation protocols in enough detail to reproduce the exact measurement.

- Research papers are inconsistent choice of evaluation setups and metrics. This seems to be caused by several factors, including a lack of consensus on best practices in the details of recommender evaluation and papers that do not specify enough details, leaving later researchers to guess.

- Because of the difficulty of understanding algorithms, algorithm authors do not always compare against high-quality implementations of prior work. For example, the baseline algorithm may lack state-of-the-art optimizations and data normalization, resulting in an evaluation that overestimates the improvement made by the proposed new approach.

Fortunately, this problem has been getting attention: 2013 saw the introduction of a RepSys workshop at the ACM Conference on Recommender Systems, discussing the challenges of recommender research reproducibility and potential solutions. There are also a number of open-source packages providing implementations of common recommendation algorithms, including SUGGEST[3], MultiLens, COFI[4], COFE, Apache Mahout[5], MyMedi-

[3] http://glaros.dtc.umn.edu/gkhome/suggest/overview/
[4] http://savannah.nongnu.org/projects/cofi/
[5] http://mahout.apache.org

aLite[6], EasyRec[7], jCOLIBRI[8], myCBR[9], PredictionIO[10], RecDB[11], and our own LensKit (chapter 3).

Closely related to reproducibility is the *usability* of research results. In a field where many research contributions are new methods for recommendation, can a practitioner take the results of a paper and incorporate them into their system? How difficult is it for a personalization engineer at a web-based business to try out the latest recommendation research in their service's discovery tools? If it is difficult to repeat and reproduce what was done in a piece of research, it is also difficult to make use of that research in practice.

Other research communities have benefited greatly from open platforms providing easy access to the state of the art. Lemur[12] and Lucene[13] provide platforms for information retrieval research. They also make state-of-the-art techniques available to researchers and practitioners in other domains who need IR routines as a component of their work. Weka [Hal+09] similarly provides a common platform and algorithms for machine learning and data mining. These platforms have proven to be valuable contributions both within their research communities and to computer science more broadly. We hope that high-quality, accessible toolkits will have similar impact for recommender systems research; so far, things are promising.

[6] http://www.ismll.uni-hildesheim.de/mymedialite/
[7] http://www.easyrec.org/
[8] http://gaia.fdi.ucm.es/projects/jcolibri/
[9] http://mycbr-project.net/
[10] http://prediction.io/
[11] http://www-users.cs.umn.edu/~sarwat/RecDB/
[12] http://www.lemurproject.org
[13] http://lucene.apache.org

2.5 Engineering Recommenders

Our vision of recommender engineering is heavily influenced by the principles of human-recommender interaction put forward by McNee, Riedl, and Konstan [MRK06b]. HRI develops and evaluates a recommender application by analyzing the application requirements in terms of the *recommendation dialogue*, *recommender personality*, and *user task*. We extend this model by explicitly acknowledging the recommendation domain as a distinct component for analysis — different algorithms may be more suitable to meeting the needs of the same type of task in different domains — and take significant, concrete steps to bring the analytic design of recommender systems closer to reality.

At the same time, McNee, Riedl, and Konstan [MRK06a] wrote of the particular problems caused by focusing myopically on recommender accuracy and ignoring other needs and criteria such as diversity and serendipity. This work has had significant impact, with a lot of research looking at non-accuracy aspects of recommender systems. There does not seem to be as much work to date advancing the broader vision of human-recommender interaction or, as we have framed it, recommender engineering. The work of Knijnenburg et al. [Kni+12] contains several significant advancements in the science needed to make HRI and recommender engineering possible; in this thesis, we focus on the tools and experiments for understanding the algorithms themselves. Algorithms and user-centered evaluation converge in the work we present in chapter 7, and further experiments and synthesis will increasingly fill in the knowledge gaps that prevent recommender systems from being directly engineered and built.

Chapter 3

Tools for Recommender Research

LensKit[1] is an open-source software package for building, researching, and learning about recommender systems. It is intended to support reproducible research on recommender systems and provide a flexible, robust platform for experimenting with different recommendation techniques in a variety of research settings.[2]

In support of these goals, LensKit provides several key facilities:

- **Common APIs** for recommendation tasks, such as *recommend* and *predict*, allow researchers and developers to build applications and experiments in an algorithm-agnostic manner.
- **Implementations of standard algorithms** for recommendation and rating prediction, making it easy to incorporate state-of-the-art recommendation techniques into applications or research.
- An **evaluation toolkit** to measure recommender performance on common data sets with a variety of metrics.
- **Extensive support code** to allow developers to build new algorithms, evaluation methodologies, and other extensions with a minimum of new work. In particular,

[1] `http://lenskit.org`

[2] This chapter is adapted and updated from material previously published by Ekstrand et al. [Eks+11]. Several members of GroupLens have contributed to this work, most significantly Michael Ludwig, Jack Kolb, and John Riedl.

```
// Load a recommender configuration (item-item CF)
LenskitConfiguration config = ConfigHelpers.load("item-item.groovy");
// Set up a data source
config.bind(EventDAO.class)
      .to(SimpleFileRatingDAO.create(new File("ratings.csv"), "\t"));

// Create the recommender
Recommender rec = LenskitRecommender.build(config);
ItemRecommender itemRec = rec.getItemRecommender();

// generate 10 recommendations for user 42
List<ScoredId> recommendations = irec.recommend(42, 10);
```

Listing 3.1: Example code to create and use a recommender.

> LensKit provides infrastructure to help developers write algorithms that integrate easily into both offline evaluation harnesses and live applications using many different types of data sources, and to make these algorithms extensively configurable.

We started LensKit in 2010 and published it in 2011 [Eks+11]. As of 2014, it consists of 44K lines of code, primarily in Java, and contains code from 12 contributors[3]. We develop LensKit in public, using GitHub[4] for source code management and bug tracking and Maven Central for distributing releases and managing dependencies.

The remainder of this chapter describes how LensKit can be used by researchers and developers, and the design and implementation that enable those uses.

3.1 Introduction to LensKit

Listing 3.1 demonstrates the basic steps that a program needs to perform in order to use LensKit to generate recommendations:

[3] Statistics from Ohloh (`https://www.ohloh.net/p/lenskit`).

[4] `https://github.com/lenskit/lenskit`

1. Configure the recommender algorithm. This is done here by loading the `item-item.groovy` configuration file, which configures an item-item collaborative filtering recommender. The LensKit documentation contains example configuration files for several different algorithms.

2. Set up a data source; in this case, tab-separated rating data from `ratings.tsv`.

3. Construct the LensKit recommender, represented in the Recommender object. This provides access to all of the facilities provided by the configured recommender.

4. Get the ItemRecommender component, responsible for producing recommendation lists for users, and use it to compute 10 recommendations for user 42.

Using and integrating LensKit revolves around a *recommender*. A LensKit recommender comprises a set of interfaces providing recommendation, rating prediction, and other recommender-related services using one or more recommender algorithms connected to a data source. These services are exposed via individual interfaces — ItemRecommender, RatingPredictor, ItemScorer, etc. — reflecting different capabilities of the recommender.

Experimenters wanting to use LensKit to compare a set of algorithms can write an evaluation script, specifying three primary things:

- The data set(s) to use
- The algorithms to test
- The metrics to use

Listing 3.2 shows simple evaluation script that will perform a cross-validation experiment on three algorithms. Experiments can be substantially more sophisticated — recording

```
trainTest {
    dataset crossfold {
        source csvfile("ratings.csv") {
            domain minimum: 1.0, maximum: 5.0, precision: 1.0
        }
    }

    metric CoveragePredictMetric
    metric RMSEPredictMetric
    metric NDCGPredictMetric

    algorithm 'pers-mean.groovy', name: 'PersMean'
    algorithm 'item-item.groovy', name: 'ItemItem'
    algorithm 'user-user.groovy', name: 'UserUser'
}
```

Listing 3.2: Example evaluation experiment.

extensive metrics over recommender models and outputs, testing procedurally generated algorithm variants, etc. — but at their core, they are measurements of algorithms over data sets. The evaluator produces its output in CSV files so it can be analyzed and charted in Excel, R, or whatever the user wishes.

3.2 Design of LensKit

We want LensKit to be useful to developers and researchers, enabling them to easily build and research recommender systems. More specifically, we have designed LensKit to be useful for building production-quality recommender systems in small- to medium-scale environments and to support many forms of recommender research, including research on algorithms, evaluation techniques, and user experience.

We also want LensKit to be useful in educational environments. As students learn how

to build and integrate recommender systems, it can be beneficial for them to use and study existing implementations and not just implement simplified versions of the algorithms. We have used it ourselves to teach a MOOC and graduate course on recommender systems [Kon+14]. However, the design and implementation been driven primarily by research and system-building considerations, and we have significant work to do in building documentation, simplified APIs, and other entry points to make it more accessible to students.

In order to turn LensKit from a concept into working code, we have needed to turn the overall project goals of supporting research and development into software architecture and finally implementations. LensKit's design and implementation are driven by a few key design principles, many of which are applications of good general software engineering practice:

Build algorithms from loosely-coupled components.

Herlocker, Konstan, and Riedl [HKR02] separates the user-user collaborative filtering algorithm into several conceptual pieces and considers the potential design and implementation decisions for each separately. We extend this principle into all our algorithm implementations: a typical recommender is composed of a dozen or more distinct components.

This decoupling achieves several important goals. First, it is good software engineering practice to separate complicated logic into distinct components that communicate via small, well-defined interfaces in order to improve maintainability, readability, and testability. An entire collaborative filtering algorithm is difficult to extensively test; item similarity functions and mean-centering normalizers can be tested with relative ease, increasing our confidence in the final system.

Second, it provides extension and configuration points to customize algorithms and

experiment with variants. Breaking the algorithm into small components is a prerequisite for allowing those components to be individually replaced and reconfigured. For example, Sarwar et al. [Sar+01] tested different item similarity functions for item-based collaborative filtering; by implementing item similarity as a distinct component in LensKit, we can conduct similar research by providing alternate implementations of the ItemSimilarity interface.

Third, it allows components to be re-used between algorithms. For instance, many algorithms benefit from normalizing user rating data prior to performing more sophisticated computations. Having distinct UserVectorNormalizer components allows us to reuse the same data normalization code across multiple algorithms.

We want researchers to be able to experiment with new algorithms or variances, evaluation metrics, etc., with a minimum of new code. Ideally, they should only need to write the code necessary to implement the particular idea they wish to try, and be able to reuse LensKit's existing code for everything else. Composing recommenders from small, replaceable building blocks is how we attempt to achieve this goal.

Be correct and safe, then efficient.

When designing components of LensKit, we naturally strive first for correct code. We also seek to design components so that the natural way to use them is likely to be correct, and so that it is difficult to violate their invariants. One result of this is extensive use of immutable objects, reducing the number of ways in which one component can break another.

To be useful, however, LensKit must also be efficient, and we have continually looked for ways to improve the efficiency of our data structures and algorithms. We also

occasionally provide means, such as fast iteration (section 3.6), for two components to negotiate a relaxation of certain assumptions in order to improve efficiency.

Use composition and the Strategy pattern, not inheritance.

Modern object-oriented programming wisdom often recommends against using inheritance as a primary means of extending and configuration code. Instead, extension points should be exposed as separate components defined by interfaces. The Strategy pattern [Gam+95b] is the foundation for this type of software design; under this scheme, if there are different ways a class could perform some portion of its responsibilities, it depends on another component with an interface that encapsulates just the reconfigurable computation instead of using virtual methods that a subclass might override. There are many benefits to this approach, two of which have significant impact on LensKit:

- Component implementations can be refactored without breaking code that configures them, so long as the strategy interface is preserved.
- It is easier to support multiple configuration points. If we had a UserUserCF class that had virtual methods for normalizing data and comparing users, configuring it would require subclassing and overriding both methods, either implementing the relevant computations or delegating to some other code that does. Composition and the Strategy pattern mean that the data normalization and user comparison algorithms can be configured by giving the user-user collaborative filter particular UserVectorNormalizer and UserSimilarity implementations, which are also provided by LensKit.

Be configurable, but have sensible defaults.

We want LensKit algorithms — and other aspects of LensKit where appropriate — to be extensively configurable, but we do not want users to have to configure (and therefore understand) every detail in order to start using LensKit. Therefore, we have broken each algorithm into many individually-configurable components (as described earlier), and continue to refactor the algorithm implementations to support more diverse configurations, but provide default component implementations and parameter values wherever sensible.

Wherever there is a sensible default, and subject to compatibility concerns, we want LensKit's default, out-of-the-box behavior to be current best practices. This is particularly true for the evaluator, where we want the result of saying 'evaluate these algorithms' to be consistent with commonly-accepted evaluation practice. LensKit's defaults will be evolving — with appropriate versioning and compatibility notices — as the research community comes to greater consensus on how to best conduct evaluations.

Minimize assumptions.

We attempt to make as few assumptions as possible about the kinds of data users will want to use LensKit with, the types of algorithms they will implement, etc. This is particularly true for low-level portions of the system, such as the data access layer; relaxing the assumptions of other aspects, such as the evaluator and various algorithm implementations, is an ongoing project.

LensKit's design has been heavily influenced by the principles in *Effective Java* [Blo08] in pursuing safe, flexible, maintainable code.

We chose Java for the implementation language and platform for LensKit for two primary reasons. First, we wanted to write it in a language that would be accessible to a wide

range programmers and researchers, particularly students; Java is widely taught and has a high-quality implementation for all common operating systems. Second, we needed a platform that provides good performance. With some care in algorithm and data structure design and coding practices, the Java virtual machine provides excellent runtime performance.

3.3 Code Organization

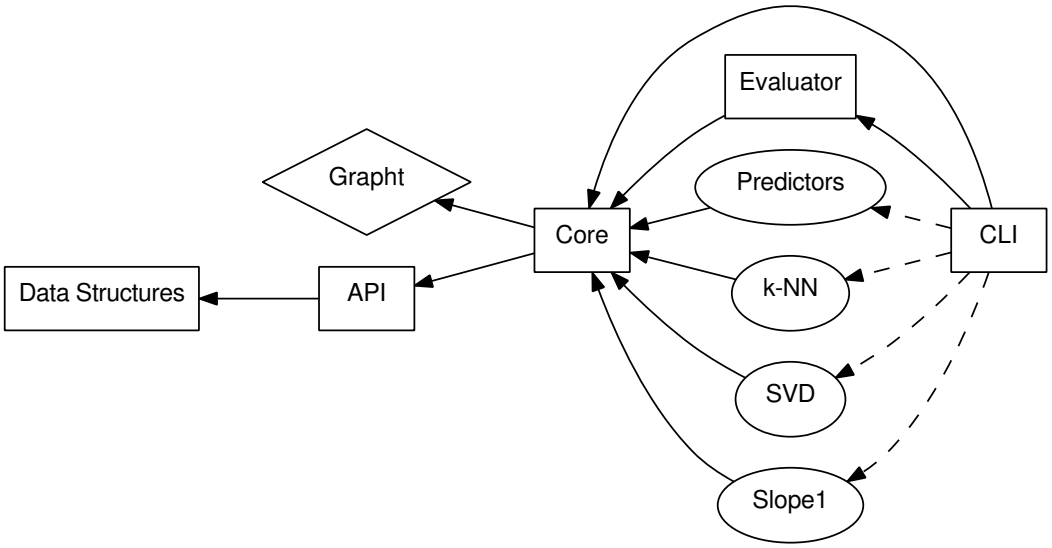


Figure 3.1: LensKit modules and their relationships

The LensKit code is divided into several modules, reflecting its design to provide lightweight common APIs and a rich support infrastructure for its algorithms, evaluators, and tools. Figure 3.1 shows the dependency relationships between these modules.

API The API module contains the interfaces comprising LensKit's recommendation API. It contains interfaces for generating recommendation lists, estimating preference, and other high-level recommendation tasks. These interfaces are independent of the rest

of LensKit (except the data structures library), so that code can be written against them and used with either LensKit's implementations or shims to expose the same interface from another toolkit such as Apache Mahout. Section 3.4 describes these APIs in more detail.

Data Structures The data structures module contains several core data structures and data-related utilities used by the reset of LensKit. Section 3.6 describes these data structures.

Core The core module contains the bulk of LensKit's except for the evaluator and algorithm implementations. It provides the support infrastructure for accessing and managing data and configuring recommender implementations, as well as baseline and default recommender components and utility classes used by the rest of LensKit.

Evaluator This module contains the LensKit evaluation tools, providing support for offline estimates of algorithm performance using widely used metrics and evaluation setups. Section 3.8 describes the evaluator.

Predictors More sophisticated rating prediction support. This includes OrdRec [KS11] and adapters for additional rating prediction.

k-NN Nearest-neighbor collaborative filtering, both user-based [Res+94] and item-based [Sar+01] algorithms.

SVD Collaborative filtering by matrix factorization; currently, the only algorithm implemented is FunkSVD [Fun06; Pat07].

Slope1 Slope One predictors for collaborative filtering [LM05].

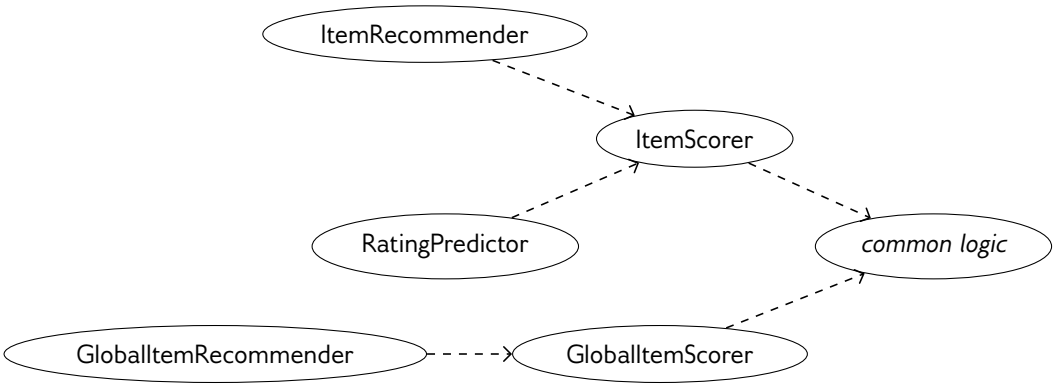


Figure 3.2: Diagram of LensKit API components

Grapht Grapht, described in more detail in chapter 4, is not technically a part of LensKit. It is the dependency injection library used by the LensKit core to configure and instantiate particular recommender algorithms.

CLI The command line interface provides tools for running LensKit evaluations, inspecting algorithm configurations, manipulating data files, etc.

3.4 Recommender APIs

The public API defined by LensKit is accessed via the Recommender interface introduced in section 3.1. Its primary implementation, LenskitRecommender, encapsulates the components that make up a particular recommender and makes them available to client applications. A Recommender does not define any particularly interesting behavior on its own; all it does is provide access to the implementations of interfaces for particular recommendation tasks. LensKit does not provide any other implementations of Recommender; it is separated from its implementation and included in the public API to provide a place to implement shims around other recommender implementations, making it possible to adapt other implementations such as Mahout to make be usable in LensKit-based applications.

Figure 3.2 shows the components that make up LensKit's public API, and how they typically interact. The central component of most LensKit recommenders is an implementation of the ItemScorer interface. The various recommendation techniques implemented by LensKit differ primarily in the item scorer they implement; in almost all cases, the algorithm to be used is configured by selecting an item scorer implementation to use (in listing 3.1, this is done inside `item-item.groovy` configuration file).

An item scorer is a generalization of the *predict* capability, computing general user-personalized scores for items. No assumptions are made or implied about what the scores mean, except that higher scores should indicate 'better' items, for some definition of 'better' that makes sense in the context of the application and algorithm. When operating on rating data, many item scorer implementations compute scores by predicting user ratings; this generalization to scores, however, allows components to operate with non-rating-based (e.g. purchase or click count data) without artificial meanings. Implementing a new algorithm for LensKit is usually done by creating a new item scorer implementation, as most algorithms are mechanisms for producing personalized scores.

Most applications embedding LensKit will not use the item scorer directly, however. Instead, they will use the RatingPredictor and ItemRecommender interfaces, providing support for the traditional *predict* and *recommend* tasks respectively.

The rating predictor and item scorer interfaces are identical (with the methods renamed from `score` to `predict`), but the contract of RatingPredictor carries the additional guarantee that its scores are interpretable as predicted ratings. Separating the item scorer and rating predictor interfaces — and the components implementing them — provides three major advantages. First, it frees up individual scorer components from dealing with some of the details of rating prediction, such as clamping ratings to the range of valid ratings and possibly quantizing them, keeping the code conceptually simple. Second, it consolidates code

for sanitizing scores to be interpretable as ratings in one place (the default RatingPredictor implementation), reducing code duplication. Third, it allows alternative strategies for mapping scores to predicted ratings, such as OrdRec [KS11], to be easily swapped in and used on top of LensKit's existing item scoring capabilities.

The item recommender interface provides lists of recommendations for a particular user in the system. The application using it provides a user ID, the desired number of recommendations n, and optionally an *candidate* set C and/or an *exclude* set E of item IDs to constrain the recommendations. The recommender will return up to n recommendations from $C \setminus E$. If unspecified, C defaults to all recommendable items and E defaults to the items the user has rated or purchased (although individual item recommender implementations may change these defaults). These sets allow the application to use LensKit in situations such as recommending from among the items in one particular category or matching some search query.

LensKit also exposes an interface GlobalItemRecommender (and an associated GlobalItemScorer for 'global' (non-personalized) recommendation that does not take the user into account, but operates with respect to zero or more items. Applications can use it to implement a 'similar items' feature or to provide recommendations based on the contents of a shopping basket.

Listing 3.3 lists the core methods exposed by several of the interfaces in the LensKit API. Section 3.7 describes many of implementations LensKit provides of these interfaces.

3.5 Data Model

LensKit recommenders need a means of accessing and representing the data — ratings, purchases, item metadata, etc. — from which they are to compute recommendations. To

```java
public interface ItemScorer {
    /**
     * Compute scores for several items for a user.
     */
    SparseVector score(long user, Collection<Long> items);
}

public interface ItemRecommender {
    /**
     * Recommend up to `count' items for a user.  Only items
     * in `candidates' but not in `excludes' are considered.
     */
    List<ScoredId> recommend(long user, int count,
                             Set<Long> candidates,
                             Set<Long> excludes);
}

public interface GlobalItemRecommender {
    /**
     * Recommend up to `count' items related to a selected
     * set of items.  Only items in `candidates' but not in
     * `excludes' are considered.
     */
    List<ScoredId> recommend(Set<Long> items, int count,
                             Set<Long> candidates,
                             Set<Long> excludes);
}
```

Listing 3.3: Simplified LensKit interfaces.

support this in a general fashion, extensible to many types of data, LensKit defines the concepts of *users*, *items*, and *events*. This design is sufficiently flexible to allow LensKit to work with explicit ratings,implicit preference extractable from behavioral data, and other types of information in a unified fashion.

Users and items are represented by numeric identifiers (Java `longs`). LensKit makes no

assumptions about the range or distribution of user and item identifiers, nor does it require users and items to have disjoint sets of identifiers. The only constraint it places upon the users and items in the data it interacts with is that they can be represented with numeric IDs.

An event is some type of interaction between a user and an item, optionally with a timestamp. Each type of event is represented by a different Java interface extending Event. Since ratings are such a common type of data for recommender input, we provide a Rating event type that represents a user articulating a preference for an item.[5] A rating can also have a null preference, representing the user removing their rating for an item. Multiple ratings can appear for the same user-item pair, as in the case of a system that keeps a user's rating history; in this case, the system must associate timestamps with rating events, so that the most recent rating can be identified.

Recommender components access the user, item, and event data through data access objects (DAOs). Applications embedding LensKit can implement the DAO interfaces in terms of their underlying data store using whatever technology they wish — raw files, JDBC, Hibernate, MongoDB, or any other data access technology. LensKit also provides basic implementations of these interfaces that read from delimited text files or generic databases via JDBC, and implement more sophisticated functionality by caching the events in in-memory data structures.

The methods these interfaces define come in two flavors. Basic data access methods, prefixed with `get` (such as `getEventsForItem(long)`), retrieve data and return it in a standard Java data structure such as a list (or a LensKit-specific extension of such a structure). Streaming methods, prefixed with `stream`, return a *cursor* of items; cursors allow client code to process objects (usually events) one at a time without reading them all into memory, and

[5] LensKit does not yet provide implementations of other event types, but it is one of our high-priority tasks.

release any underlying database or file resource once processing is completed or abandoned.

The standard LensKit DAO interfaces are:

EventDAO The base DAO interface, providing access to a stream of events. Its only methods are to stream all events in the database, optionally sorting them or filtering them by type.

ItemEventDAO An interface providing access to events organized by item. With this interface, a component can retrieve the events associated with a particular item, optionally filtering them by type. It can also stream all events in the database grouped by item.

UserEventDAO Like ItemEventDAO, but organized by user.

ItemDAO An interface providing access to items. The base interface provides access to the set of all item IDs in the system.

UserDAO An interface providing access to users. Like ItemDAO, it provides access to the set of all user IDs in the system.

An application that augments LensKit with components needing additional information, such as user or item metadata for a content-based recommender, will augment these interfaces with additional interfaces (possibly extending the LensKit-provided ones) to provide access to any relevant data. We have done this ourselves when embedding LensKit in an application or using it for an experiment; for example, in teaching our recommender systems MOOC, we extended ItemDAO with methods to get the tags for a movie to allow students to build a tag-based recommender in LensKit.

Early versions of LensKit had a single DataAccessObject interface that was handled specially by the configuration infrastructure; it was possible to extend this interface to pro-

vide extra data such as tags, but it was not very easy. Since LensKit 2.0, the data access objects are just components like any others, and receive no special treatment.

3.6 Data Structures

LensKit implements several data structures and data-related utilities to support building and working with recommenders.

There are many places where we need to be able to manipulate vectors of values associated with users and items, such as a user rating vector containing the user's current rating for each item they have rated. To support these uses, LensKit provides a *sparse vector* type. Sparse vectors are optimized maps from `longs` to `doubles` with efficient support for linear algebra operations such as dot products, scaling, and addition. Initially, we tried using hash maps for these vectors, but they performed poorly for common computations such as vector cosines.

The SparseVector class uses parallel arrays of IDs and values, sorted by ID. This provides memory-efficient storage, efficient ($O(\lg n)$) lookup by key, and enables many two-vector operations such as dot products to be performed in linear time by iterating over two vectors in parallel. This class helps LensKit algorithm implementers write many types of algorithms in a concise and efficient manner. Sparse vectors also provide type-safe immutability with three classes: the abstract base class SparseVector provides the base implementation and read-only methods; ImmutableSparseVector extends it and guarantees that the vector cannot be changed by any code; and MutableSparseVector extends SparseVector with mutation operations such as setting or adding individual keys or sets of keys from another vector.

To maintain predictable performance, the sparse vectors do have one key limitation:

when created, a sparse vector has a fixed *key domain*, the set of all keys that can possibly be stored in it. Individual entries can be set or unset, but once a sparse vector (even an immutable one) is created, no entry can be added whose key was not in the original key domain. This means that sparse vectors never have to reallocate or rearrange memory: getting or setting the value for a key is either $O(\lg n)$ or fails in all cases. Programmers using sparse vectors must organize their code to work around this, setting up the list of keys they need in advance In practice, most code we have written can easily know in advance the set of user or item IDs that it will need to work with and allocate a vector without incurring overhead in either run time or code bloat. For those cases where the IDs are discovered on-the-fly, we use a more dynamic structure such as a hash map and convert it to a vector when we are finished.

LensKit also provides additional data structures for associating lists of events with user or item IDs, mapping long IDs to contiguous 0-based indexes (helping to store user or item data in arrays), and associating scores with IDs either on their own or in lists (where the sorted-by-key property of sparse vectors is undesired).

In addition to its own data structures, LensKit makes heavy use of fastutil[6] and Google Guava. The fastutil library provides primitive collections that are compatible with the Java collections API, allowing LensKit to have lists, sets, and maps of unboxed longs and doubles. We use these extensively throughout the LensKit code to reduce memory consumption and allocation overhead, significant sources of slowdown in naïve Java code.

LensKit also borrows the *fast iteration* pattern from fastutil for its own data structures; under fast iteration, an iterator can mutate and return the same object repeatedly rather than returning a fresh object for each call to its `next()` method. For classes that present flyweights over some internal storage (e.g. entry objects representing key-value pairs in a

[6] `http://fastutil.dsi.unimi.it/`

sparse vector), this can significantly reduce object allocation overhead. Reducing needless object allocations has resulted in many significant, measurable performance improvements. Many LensKit data structures and the Cursor API support fast iteration.

3.7 Modular Algorithms

In order to reproduce a wide variety of previously-studied algorithms and configurations, as well as facilitate easy research on new configurations and tunings of existing recommender algorithms, LensKit uses a heavily modular design for its algorithm implementations. LensKit also provides configuration facilities built around this design to make it easy to configure, instantiate, and use modular algorithms.

LensKit algorithms are, wherever practical, broken into individual components that perform discrete, isolated portions of the recommendation computation, as discussed in section 3.2. Similarity functions, data normalization passes, baseline predictors, and neighborhood finders are just some examples of the types of distinct components in LensKit algorithms. The Strategy pattern [Gam+95b] provides the basis for the design of many of these components and their interactions.

We also make significant use of builders or factories. We prefer to create immutable components (or at least visibly immutable objects — some have internal caching mechanisms), and keep components that exist primarily to make data available simple. To that end, we will make a component that is a data container with well-defined access operations paired with a builder to do the computations needed to build the object. This keeps the build computations separate from the (relatively simple) access operations, and also allows the build strategy to be replaced with alternative strategies that produce the same type of data object.

Together, the modularity and separation strategies LensKit employs provide two significant benefits:

- Algorithms can be customized and extended by reimplementing just the components that need to be changed. For example, if a researcher wishes to experiment with alternative strategies for searching for neighbors in user-based collaborative filtering, they only need to reimplement the neighborhood finder component and can reuse the rest of the user-based CF implementation.

- New algorithms can be built with less work by reusing the pieces of existing algorithms. A new algorithmic idea that depends on item similarity functions and a transposed (item-major) rating matrix can reuse those components from the item-item CF implementation.

LensKit algorithms also make significant use of other common design patterns, such as Builder and Facade, to organize algorithm logic [Gam+95a].

To enable components to be configurable, LensKit components are designed using the *dependency injection* pattern [Mar96]. The idea of dependency injection is that a component requires the code that instantiates it to provide the objects on which it depends rather than instantiating them directly. With this design, the caller can substitute alternate implementations of a component's dependencies and substantially reconfigure its behavior.

Since a LensKit algorithm implementation consists of many interoperating components, instantiating all the objects needed to use one is cumbersome, error-prone, and difficult to keep up-to-date as the code is extended and improved. LensKit uses an automated dependency injector (Grapht; see chapter 4) to ease this process. Grapht scans the Java class implementing each component, extracts its dependencies, and instantiates the graph of ob-

jects needed to realize a particular recommendation algorithm. As shown in listing 3.1, the client code just needs to specify what implementation it wants to use for various interfaces, and the values of some parameters, and LensKit will use Grapht to instantiate the objects correctly. Figure 3.3 shows the full object diagram for LensKit's item-item collaborative filter; the large number of components intricate dependency edges would be impractical to instantiate manually. LensKit also provides defaults for most of its interfaces and parameters, so users only need to specify configuration points where they wish to deviate from the default.

This results in such things as being able to use item-item collaborative filtering as a baseline for a matrix factorization approach. We have sought to avoid imposing artificial limits on the ways that components can be combined.

The components in a LensKit algorithm generally divide into two categories: *pre-built* components are built once from the underlying data and can be reused across multiple recommender invocations; they may go a bit stale in a production system, but are usually rebuilt on a schedule (e.g. nightly) to take new data into account. These components are often statistical models, precomputed matrices, etc.; they are marked with the `@Shareable` annotation to allow LensKit's tooling to recognize and work with them. *Runtime* components need live access to the data source and are used to directly produce recommendations and predictions. This distinction is used to aid in web integration (section 3.9) and speed up evaluation (section 3.8.4).

3.7.1 Basic Component Implementations

LensKit provides basic implementations of several of its core interfaces. The net effect of these implementations and LensKit's default settings is that the user only needs to configure the item scorer to get a reasonably full-featured recommender that can generate recommen-

dation lists and predict ratings. They also provide common functionality to enable new recommenders to be written without extensive code duplication.

The TopNItemRecommender uses the item scorer to score the candidate items and recommends the top N. It is the default implementation of ItemScorer. By default, it excludes the items that the user has interacted with (e.g. rated), unless the client specifies a different exclude set.

SimpleRatingPredictor implements RatingPredictor by wrapping an ItemScorer and clamping the item scores to be within the range of allowable ratings. It just does a hard clamp of the values without any other rescaling. If no rating range is specified, the rating predictor passes through the item scores unmodified. Integrators can specify rating ranges by configuring a PreferenceDomain object.

The simple rating predictor can also use a second item scorer, the *baseline scorer*, specified with the @BaselineScorer qualifier. If a baseline scorer is available, it is used to supply scores for items that the primary scorer cannot score. Most recommenders are configured to use a full recommendation algorithm as the primary scorer and a simple but always-successful scorer, such as a personalized mean or item average, as the baseline scorer, so the system can always predict ratings. LensKit also provides a FallbackItemScorer that implements the fallback logic as an item scorer instead of a rating predictor; this can be used to allow other components using an item scorer, such as an item recommender, to use the fallback scores.

LensKit also has a QuantizedRatingPredictor that rounds the scores produced by an item scorer to the nearest valid rating value (e.g. on a half-star rating scale, it will round them to the nearest 0.5).

3.7.2 Summarizers and Normalizers

Many recommender algorithms have historically operated on the user's rating vector. Systems that do not use explicit ratings may produce some kind of a vector over items for each user, representing the user's history with or preference for that item, such as a vector of play or purchase counts. Several algorithms can be adapted to implicit preference data simply by using some vector other than the rating vector, perhaps with small tweaks to the algorithm's computations.

In both implicit and explicit cases, it is also common to normalize the vector in some way, such as mean-centering it normalizing it to z-scores.

LensKit exploits this potential for generalizability with *history summarizers*, expressed in the UserHistorySummarizer interface. A history summarizer takes a user's history, expressed as a list of events, and produces a sparse vector whose keys are items and values are some real-valued summary of the user's preference for that item.

The default history summarizer is RatingVectorUserHistorySummarizer, which summarizes a user's profile by extracting their most recent rating for each item. There is also EventCountUserHistorySummarizer, which counts the number of times some type of event occurs for each item. Using this summarizer with an event type Play, for example, would count the number of Play events associated with each item for that user, resulting in a play count vector.

For convenience, in the remainder of this section, we will use *rating* to refer to the summarized value associated with an item in the user's profile; if we need to distinctly refer to different types of events, we will call them Rating events.

As well as summarizers, LensKit provides and uses various *normalizers*. The most general normalizers are vector normalizers, defined by the VectorNormalizer interface. Vector

```
SparseVector userData = /* user ratings */;
// capture a transformation based on the user data
VectorTransformation xform = normalizer.makeTransformation(userData);
// normalize the user's data
MutableSparseVector normed = userData.mutableCopy();
xform.apply(normed);
MutableSparseVector output = /* construct output vector */;
// do some computations to produce outputs
// and de-normalize the data at the end:
xform.unapply(output);
```

Listing 3.4: Use of vector normalizers.

normalizations operate on two vectors: the *reference* vector and *target* vector. The reference vector is used to compute the basis of the normalization; for example, MeanCenteringVectorNormalizer computes the mean of the reference vector. The target vector is the vector actually modified. If they are the same vector, normalization has the effect of e.g. subtracting the mean value from every value in the vector.

To be reversible, vector normalizers also support creating a transformation from a reference vector. This operation captures the transformation that will be applied, such as the mean value to subtract. The transformation can then be applied and unapplied to any vector. Listing 3.4 shows this in action: normalizing user data, computing some output, and then using the original transformation (such as a mean-centering transform using the user's mean rating) to de-normalize the output.

In addition to the generic VectorNormalizer interface, LensKit provides user- and item-specific normalizers. These interfaces function identically to the vector normalizer, including producing vector transformations, except that they take a user or item ID in addition to a reference vector. This allows normalizations to take into account other information about the user or item; such normalizers often depend on either a DAO to access user or item data,

or some other component which may take advantage of knowing the user or item for which a vector should be normalized. The default implementations of these interfaces ignore the user or item ID and delegate to a VectorNormalizer.

3.7.3 Baseline Scorers

LensKit provides a suite of *baseline* item scorers, using simple averages to compute scores for items. These baselines serve multiple roles. They are often used as fallbacks to predict ratings or compute recommendations when a more sophisticated recommender cannot (e.g. a nearest-neighbor collaborative filter cannot build a neighborhood). They are also used for data normalization — many standard algorithm configurations apply the sophisticated algorithm to the residual of the baseline scores rather than the raw ratings. This is done by using the baseline-subtracting user vector normalizer, an implementation of UserVectorNormalizer (section 3.7.2) that transforms vectors by subtracting the score produced by a baseline scorer. This paradigm is an extension of the mean-centering normalization that has long been employed in recommender systems and other data analysis algorithms. It results in the following final scoring rule [ERK10], where b_{ui} is the baseline score for user u and item i:

$$\text{score}(u, i) = b_{ui} + \text{score}'(u, i)$$

All baseline scorers implement the ItemScorer interface, so they can be used on their own to score items. This also means that any item scorer can be used as a baseline in another algorithm, a significant aspect of the composability of LensKit algorithms. LensKit provides the following baseline scorers:

ConstantItemScorer Scores every item with some pre-defined constant value.

GlobalMeanRatingItemScorer Scores every item with the global mean rating ($b_{ui} = \mu$).

ItemMeanRatingItemScorer Scores every item with its mean rating, so $b_{ui} = \mu_i$. This scorer also takes a damping parameter γ to bias the computed mean ratings towards the global mean for items with few ratings. A small number of ratings is not a good sample of the true quality of that item; this term keeps items from having extreme means without substantial evidence to support such values. Equations (3.1) and (3.2) show the full formulas for this scorer, with the set U_i consisting of users who have rated item i:

$$\bar{\mu}_i = \frac{\sum_{u \in U_i} (r_{ui} - \mu)}{|U_i| + \gamma} \tag{3.1}$$

$$b_{ui} = \mu + \bar{\mu}_i \tag{3.2}$$

$\gamma > 0$ is equivalent to assuming *a priori* that every item has γ ratings equal to the global mean μ. When few users rate the item, it damps the effect of their ratings so that the system does not assume that an item has 5 stars because it has a single 5-star rating. As more users rate the item, the real ratings increasingly dominate these fake ratings and the baseline score approaches the simple mean of user ratings for the item.

UserMeanItemScorer This scorer is more sophisticated. It depends on another scorer (designated the *user mean baseline*), producing scores b'_{ui}, and a user history summarizer that produces a vector $\vec{u}$ of item values for the user (e.g. ratings). It computes the mean difference $\hat{\mu}_u$ between the user's value for each item and the user mean baseline's scores for that item; it scores each item with its baseline score and the user mean offset. Equations (3.3) and (3.4) show the formulas for this computation; γ is

again a damping term, working as it does in the item mean rating scorer, and I_u is the set of items in $\vec{u}$.

$$\hat{\mu}_u = \frac{\sum_{i \in I_u} (u_i - b'_{ui})}{|I_u| + \gamma} \tag{3.3}$$

$$b_{ui} = b'_{ui} + \hat{\mu}_u \tag{3.4}$$

If the user has no ratings, $\hat{\mu}_u = 0$, so this scorer's scores fall back to the underlying baseline. If the item mean rating scorer is used as the user mean baseline and user profiles are summarized by rating vectors, then $\hat{\mu}_u$ is the user's average deviance from item average rating and $b_{ui} = \mu + \bar{\mu}_i + \hat{\mu}_u$. If the global mean rating is used as the baseline, then this scorer uses the user's average, falling back to the global mean when the user has no ratings.

3.7.4 Item-Item CF

LensKit components are also designed to be as composable as practical, so they can be combined in arbitrary fashions for maximal flexibility. LensKit's item-item CF implementation provides item-based collaborative filtering over explicit ratings [Sar+01] (including rating prediction) and implicit data represented as item preference vectors [DK04]. Item-based CF examines a user's profile and recommends items similar to items they have liked in the past.

The item-item implementation consists of many components; fig. 3.3 shows a typical configuration as specified in listing 3.6. The core of the item-item implementation is ItemItemScorer, an implementation of ItemScorer using item-based CF. It combines a user's preference data, computed using a summarizer, with item neighborhoods provided

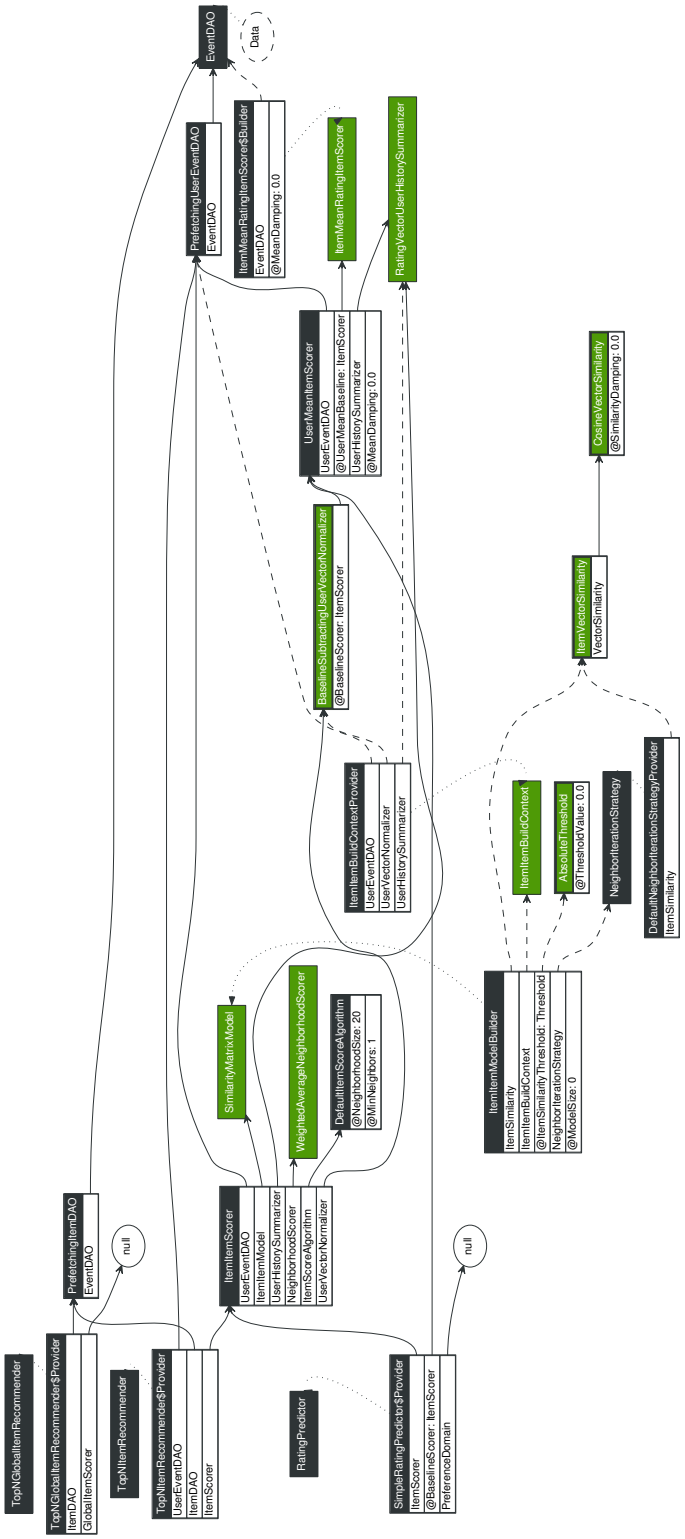


Figure 3.3: Diagram of the item-item recommender.

by an ItemItemModel. Each neighborhood consists of a list of items with associated similarity scores, sorted in nonincreasing order by similarity.

Equation (3.5) shows the basic formula for LensKit's item-item collaborative filter, where $\vec{b}_u$ is the baseline scores for each of the items in the user's summary vector $\vec{u}$, $N(i)$ is the neighborhood of i, and f is a neighborhood scoring function. Typically, $N(i)$ is limited to the n items most similar to i that also appear in $\vec{u}$. If the normalizer g is a baseline-subtracting normalizer, the formula expands to eq. (3.7).

$$\text{score}(u,i) = g^{-1}(f(i, N(i), g(\vec{u}))) \tag{3.5}$$

$$g(\vec{r}) = \vec{r} - \vec{b}_u \tag{3.6}$$

$$\text{score}(u,i) = b_{ui} + f(i, N(i), \vec{u} - \vec{b}_u) \tag{3.7}$$

Computing item scores from a neighborhood and the user vector is abstracted in the NeighborhoodScorer component (f in eq. (3.5)). There are two primary implementations of this interface: WeightedAverageNeighborhoodScorer computes the average of the user's ratings or scores for each item in a neighborhood, weighting them by the item's similarity to the target item:

$$f(i, N, \vec{u}') = \frac{\sum_{j \in N} \text{sim}(i,j) u'_j}{\sum_{j \in N} |\text{sim}(i,j)|}$$

SimilaritySumNeighborhoodScorer simply sums the similarities of all items that the appear in the user's summary; this is useful for unary domains such as purchases where the user summary value is 1 for items the user has purchased and 0 otherwise.

The default ItemItemModel is SimilarityMatrixModel, which stores a list of neighbors for each item in memory. It is not a full matrix, but rather a mapping from item IDs to neigh-

bor lists forming a specialized sparse matrix. The similarity matrix model in turn is built by the ItemItemModelBuilder. The item-item model depends on two primary components: the item similarity function, specified by the ItemSimilarity interface, and an ItemItemBuildContext. The build context consists of the ratings matrix, normalized and organized by item. It is implemented as a map of item IDs to sparse vectors of user ratings for that item. With this separation, the model builder only needs to compute and store item similarities, and the build context builder takes care of whatever normalization or other data pre-processing is needed.

The ItemItemModelBuilder can take advantage of 2 properties of the similarity function: whether or not it is symmetric, and whether or not it is sparse. Symmetric similarity functions have their ordinary definition: a function is symmetric iff $s(i, j) = s(j, i)$. Sparse similarity functions are functions that will return 0 if the items have no users in common. Most common similarity functions, such as vector cosine, are both sparse and symmetric. Conditional probability [DK04] is a notable example of a non-symmetric similarity function, and item similarity functions that take into account external data such as item metadata may be non-sparse. The ItemSimilarity interface has `isSparse()` and `isSymmetric()` methods to allow a similarity function to report its behavior.

If the similarity function is symmetric, the default model builder takes advantage of that by only computing the similarity between each unordered pair of items once and storing each item in the other's neighborhood. If the similarity function is sparse, then (by default) the model builder will attempt to exploit that sparsity to reduce the number of item comparisons it actually makes. In addition to the item-indexed rating matrix, the build context contains a mapping from user IDs to sets of item IDs that they have rated. For each row in the similarity matrix it is building, the model builder iterates over the users that have rated the row's item, then over each of that users' items, skipping items it has already seen. For items

with few potential neighbors, this can greatly reduce the number of comparisons that need to be performed. The sparsity exploitation is also adaptive: if, while scanning the potential neighbors for a row, the model builder gets to a point where it has processed 75% of the items but still has at least 50% of the row's users left to go, it skips the sparsity and just compares with the rest of the items.[7] In this way, it attempts to avoid situations where the bookkeeping for exploiting sparsity is more expensive than the extra item comparisons; this can happen when the data set has relatively few items compared to the number of users. This capability can also be disabled completely if it is causing problems for a particular data set or experiment.

The model builder also takes a few additional parameters. @ModelSize controls the maximum number of neighbors retained for each item. The model builder keeps a size-limited heap for each item, allowing it to efficiently retain the most similar neighbors for each item. If the model size is 0, the model builder retains all neighbors.

Finally, the model builder takes a Threshold to filter neighbors. The default threshold excludes all neighbors with a negative similarity. This allows the neighborhoods to be filtered to require items to have some minimum similarity (or minimum absolute similarity). Neighbors are filtered before being counted, so a model size of 500 retains the 500 most similar items that pass the threshold.

The item-item CF implementation also contains two variants on the basic model building process. The NormalizingItemItemModelBuilder replaces ItemItemModelBuilder and allows item neighborhood vectors to be normalized. Rather than accumulating item neighborhoods in a map of heap-backed accumulators, it processes items strictly one at a time, declining to take advantage of symmetry or sparsity. After computing all the similarities in

[7] These values have not been empirically tuned, but seem to work reasonably well in practice in our applications.

each row, it applies an ItemVectorNormalizer to that row's vector. This allows techniques such as normalizing an item's neighborhood to the unit vector [DK04]. The model builder finally truncates the neighborhood with a size cap and/or a threshold and moves on to the next item.

The default item-item build context builder processes the input data on a user-by-user basis, summarizing the user's profile and applying a UserVectorNormalizer to the summary prior to storing each item value in its corresponding item vector. We have found, however, that centering user ratings by item mean is an effective normalization strategy [Eks+11]; processing ratings user-by-user and then storing them by item is needlessly memory- and time-intensive for this simple strategy. The ItemwiseBuildContextBuilder replaces the default build context builder and processes ratings item-by-item, applying a ItemVectorNormalizer to each item's rating vector. The MeanCenteringVectorNormalizer can be used for normalizing the item vectors normalize them by subtracting the item's average rating from each rating. This context builder reduces both the memory and time requirements of the item-item model build process in many situations, at the expense of supporting per-user normalization and general summarizers (it only considers ratings).

To summarize:

1. The ItemItemBuildContextBuilder summarizes the user profiles, normalizes them, and builds the ItemItemBuildContext. The build context is a rating-indexed matrix of user-item preference measurements.

2. The ItemItemModelBuilder uses the context and the ItemSimilarity to build an item-item similarity matrix. The similarity matrix implements ItemItemModel.

3. The default ItemSimilarity implementation ignores the item IDs and delegates to a VectorSimilarity.

4. The ItemItemScorer uses the ItemItemModel, the UserHistorySummarizer, and the UserEventDAO to obtain the user's current profile and score items by their similarity to items the user likes. This is done using the NeighborhoodScorer, for which there are different implementations for aggregation algorithms appropriate for different types of input data.

The LensKit-provided components have the following configuration points:

- The context preparation strategy (the provider for ItemItemBuildContext).
- The user vector summarizer.
- The user vector normalizer (used both for pre-processing data for the context and for normalizing and denormalizing rating in the scoring process; it is possible to configure these separately, but usually results in bad performance).
- For itemwise context preparation, the item rating vector normalizer.
- The model building strategy (default or normalizing).
- For the normalizing model building strategy, the item neighborhood vector normalizer.
- The maximum number of neighbors to retain for each item (@ModelSize).
- The item similarity function.
- The neighborhood score aggregation algorithm.
- @NeighborhoodSize, the maximum number of items to use when computing each score.

In addition to supporting most standard configurations of item-item collaborative filtering, this setup allows a great deal of flexibility for novel adaptations as well. While we have not yet done so, it would not be difficult to incorporate ranking metrics [AMT05; Eks+10] into the model building process. In other experiments (and a homework assignment for our recommender systems course), we have completely replaced the item-item model with one that returns neighborhoods based on a Lucene index of movie tags [ER12]. LensKit's item-item CF implementation has extensive flexibility while still performing well on common sizes of data sets.

3.7.5 User-User CF

User-based collaborative filtering, first presented by Resnick et al. [Res+94], is the oldest form of modern automated collaborative filtering. Unlike item-based CF, it finds users who are similar to the active user and recommends things liked by those users. This is typically done by using user neighborhoods to estimate the active user's preference for each item, often with a weighted average of neighbors' ratings, and recommending the top-predicted items.

LensKit's user-user CF implementation is also extensively modular, although it does not have a concept of a model. The user-user CF code is currently limited to using explicit ratings; there are no fundamental problems we know of that would prevent it from being extended to arbitrary user summaries, we just have not yet written that code.

The central class is UserUserItemScorer; selecting it as the implementation of ItemScorer will result in a user-user collaborative filter. The user-user item scorer uses a UserEventDAO to get user data, a UserVectorNormalizer to normalize user data (both that of the active user and their potential neighbors), and a NeighborFinder to find user neighbors.

The neighborhood finder has a single method that takes a user profile and a set of items

that need to be scored. It returns a collection of potential neighbors, each neighbor object representing a user with its rating vector and similarity to the active user. The scorer uses these neighbors to score the items as shown in eq. (3.8); $\tilde{u}$ denotes the normalized version of a user u or a rating value. If users are normalized by mean-centering their ratings, this equation reduces to the same formula as used by Resnick et al. [Res+94].

$$\text{score}(u, i) = \text{denorm}\left(\frac{\sum_{v \in N(u,i)} \text{sim}(\tilde{u}, \tilde{v})\tilde{r}_{vi}}{\sum_{v \in N(u,i)} |\text{sim}(\tilde{u}, \tilde{v})|}; u\right) \tag{3.8}$$

The default implementation of the neighborhood finder scans the event database for potential neighbors. Only those users who have rated one of the items to be scored are useful as neighbors; further, with a sparse similarity function, users who have not rated any of the same items as the active user will not be good neighbors. To optimize the search, the neighborhood finder takes the smaller of the active user's set of rated items and the set of target items, and considers all users who have rated at least one item among them.

LensKit also includes a neighborhood finder that uses the same logic but with a snapshot of the rating data stored in memory as a 'model'. This is much more efficient to access than a database for finding neighbors and makes user-user a more practical algorithm. When using this neighborhood finder, the active user's most recent ratings are still used, but their potential neighbors are considered frozen in time as of the last time a snapshot was taken. In production, this would likely happen nightly.

LensKit supports a full range of user similarity functions via the UserSimilarity interface. This interface is equivalent to the ItemSimilarity interface of item-item CF, and generally delegates to a VectorSimilarity.

3.7.6 Matrix Factorization CF

LensKit provides biased matrix factorization based on the work of Funk [Fun06] and subsequent developments [Pat07]. Biased matrix factorization takes the rating matrix $\mathbf{R}$, subtracts the baseline scores (often user and item biases), and computes a factorization resembling a singular value decomposition:

$$\mathbf{R} = \mathbf{B} + \mathbf{W}\mathbf{\Sigma}\mathbf{X}^{\mathrm{T}}$$

There are various ways of computing the decomposition of the matrix; LensKit's factorization architecture allows for different factorization algorithms to be plugged in and defaults to using gradient descent to learn user-feature and item-feature matrices [Fun06].

LensKit's matrix factorization package consists of two main parts. The general matrix factorization package provides components for using a biased matrix factorization independent of how it was computed:

MFModel A generic matrix factorization model, exposing the user- and item-feature matrices. It also stores mappings between user and item IDs and their respective row and column numbers in the matrices. It does not separate out the Σ matrix of singular values (feature weights); instead, they are folded into the user and item matrices.

BiasedMFKernel An interface for kernel functions to recombine user- and item-feature vectors to produce a prediction. It takes the baseline score, user vector, and item vector, and produces a final user-item score. The default implementation, DotProductKernel, adds the dot product of the vectors to the baseline score:

$$\text{score}(u, i) = b_{ui} + \mathbf{w}_{(u)} \cdot \mathbf{x}_{(i)}$$

An alternate implementation, DomainClampingKernel, operates like the dot product kernel but clamps the value to be within the valid range of ratings after each addition (numbering features 1 through $f_{\max}$) [Fun06]:

$$\text{score}(u,i) = s(u,i,f_{\max})$$

$$s(u,i,f) = \begin{cases} b_{ui} & f = 0 \\ \text{clamp}(s(u,i,f-1) + w_{uf}x_{if} & f > 0 \end{cases}$$

BiasedMFItemScorer Uses a baseline scorer, MFModel, and BiasedMFKernel to score items for users, implementing the ItemScorer interface.

The general biased MF classes can produce scores, but have no way to learn the model. The regularized gradient descent (FunkSVD) classes fill in this gap and provide some additional functionality. FunkSVDModelBuilder builds a FunkSVDModel (a subclass of MFModel) using gradient descent over the ratings in the system. It learns the features one at a time, training each to convergence before moving on to the next; this iteration is controlled by a learning rate λ, a regularization coefficient γ, and a stopping condition (usually an epoch count or a threshold). Listing 3.5 shows the algorithm for training the model.

The factorization produced by this algorithm is not a well-formed singular value decomposition. It does not have a distinct $\mathbf{\Sigma}$ matrix, but that can be extracted from $\mathbf{W}$ and $\mathbf{X}$ by setting $\sigma_f = \|\mathbf{w}_{(f)}\|_2\|\mathbf{x}_{(f)}\|_2$. More importantly, the left and right matrices do not form an orthogonal basis as they do in a true SVD. As a result, standard SVD and latent semantic analysis techniques such as computing updated user and item vectors by 'folding in' [BDO95; Sar+02] do not operate correctly.

The base MF item scorer, BiasedMFItemScorer, does not do any updating of user or item vectors: if the user does not have a feature vector in the model, it just returns the

```
1:  procedure TrainMF(R, k)
2:      shuffle list of ratings
3:      W ← new m × k matrix filled with 0.1
4:      X ← new n × k matrix filled with 0.1
5:      for f ← 1 to k do
6:          repeat
7:              for rating r_{u,i} in R do
8:                  p_{u,i} ← b_{u,i} + Σ_{k=1}^{f} u_{u,k} m_{i,k}
9:                  ϵ ← r_{u,i} − p_{u,i}
10:                 w_{u,k} ← w_{u,k} + λ(ϵ x_{i,k} − γ w_{u,k})
11:                 x_{i,k} ← x_{i,k} + λ(ϵ w_{u,k} − γ x_{i,k})
12:         until feature f converges
13:     return W, X
```

Listing 3.5: FunkSVD training algorithm.

baseline scores. FunkSVDItemScorer is more sophisticated: it can take the active user's current rating vector and do a few rounds of gradient descent to freshen up their feature vector prior to computing scores. If the user is new and does not have a feature vector in the model, it will use the average user weight for each feature as the starting point.[8]

For efficient iteration, the FunkSVDItemBuilder uses a helper structure, a PackedPreferenceSnapshot, to represent the data over which it is to train. Currently, packed preference snapshots are built from rating data directly, but alternative means of building them would allow FunkSVD to operate on other types of data without further changes.

3.7.7 Configuring Algorithms

On top of Grapht's configuration API (section 4.4.8), LensKit provides a simple syntax for configuring recommender algorithms. This syntax is implemented as an embedded domain-specific language in Groovy, a popular scripting language for the Java virtual machine with

[8] We do not yet have a lot of experience using this code in production, so it is not well-tested and its behavior is not yet well understood.

```
import org.grouplens.lenskit.baseline.*
import org.grouplens.lenskit.transform.normalize.*
import org.grouplens.lenskit.knn.*
import org.grouplens.lenskit.knn.item.*

bind ItemScorer to ItemItemScorer
bind (BaselineScorer, ItemScorer) to UserMeanItemScorer
bind (UserMeanBaseline, ItemScorer) to ItemMeanRatingItemScorer
bind UserVectorNormalizer to BaselineSubtractingUserVectorNormalizer
```

Listing 3.6: Item-item configuration file (producing the setup in fig. 3.3).

good facilities for building fluent syntaxes.

The syntax is a relaxed version of the Grapht API with some scoping conveniences for managing context-sensitive configuration. Listing 3.6 shows an example configuration of an item-item collaborative filter as a Groovy script.

This syntax provides a more syntactically lightweight means of configuring recommenders than full Java syntax. It also allows recommender definitions to be treated as configuration files rather than embedded in the source files of an application, and the LensKit command line tools operate on these scripts.

3.8 Offline Evaluation

LensKit's evaluation toolkit provides support for running offline, data-driven evaluations of recommender performance using a traditional train/test approach with cross-validation. We intend it to be a versatile platform for reproducible recommender evaluations and experiments; in our own work, we generally publish the evaluation scripts used to produce our results [Eks+11; ER12], and we encourage others to do the same.

The following goals drove the design of the LensKit evaluator:

- Writing evaluations should be easy, with minimal cumbersome syntax.
- Best practices should be the default.
- The toolkit should be flexible enough to reproduce a wide range of experiments, including those with flawed methodologies, and experiment with new evaluation techniques.
- Evaluations should be as efficient as possible.
- It is not LensKit's job to analyze the results of the evaluation. Evaluation output (metrics, actual predictions and recommendations, etc.) should be written to CSV files for further analysis with R, Excel, SciPy, or other software.

The LensKit evaluator provides facilities for processing data sets (crossfold splitting, subsampling, format conversion) and evaluating algorithms over multiple train-test data sets and measuring their performance.

3.8.1 Evaluation Scripts

LensKit evaluations are defined with Groovy scripts, using an embedded DSL to describe different types of evaluation actions. Evaluations are organized around *tasks*, such as `crossfold` (to partition data for cross-validation) and `trainTest` (to run a train-test evaluation over one or more data sets). Tasks can optionally be contained within *targets*; in this case, their execution is deferred until the target is executed, allowing a single evaluation script to define multiple evaluation capabilities.[9] Listing 3.7 shows an example evaluation script of two algorithms over the MovieLens 100K data set.

[9]The 'target' functionality will be deprecated in the future, as we plan to simplify the LensKit evaluator to be controlled by the Gradle build system instead of implementing its own build logic.

```
// imports elided
trainTest {
  dataset crossfold("ml-100k") {
    source csvfile("ml-100k/u.data") {
      delimiter "\t"
      domain minimum: 1.0, maximum: 5.0, precision: 1.0
    }
    holdout 10
    partitions 5
  }

  output "eval-results.csv"
  metric CoveragePredictMetric
  metric RMSEPredictMetric
  metric NDCGPredictMetric

  algorithm("ItemItem") {
    // use the item-item rating predictor with a baseline and normalizer
    bind ItemScorer to ItemItemScorer
    bind (BaselineScorer, ItemScorer) to UserMeanItemScorer
    bind (UserMeanBaseline, ItemScorer) to ItemMeanRatingItemScorer
    bind UserVectorNormalizer to BaselineSubtractingUserVectorNormalizer

    set ModelSize to 500
    set NeighborhoodSize to 30
  }

  algorithm("UserUser") {
    // use the user-user rating predictor
    bind ItemScorer to UserUserItemScorer
    bind (BaselineScorer, ItemScorer) to UserMeanItemScorer
    bind (UserMeanBaseline, ItemScorer) to ItemMeanRatingItemScorer
    bind VectorNormalizer to MeanVarianceNormalizer
    within(NeighborhoodFinder) {
      bind VectorNormalizer to MeanCenteringVectorNormalizer
    }

    set NeighborhoodSize to 30
  }
}
```

Listing 3.7: Example of a LensKit evaluation script.

Since these scripts are written in a full programming language, researchers have a great deal of flexibility in the configurations they can generate. For example, it is common to generate algorithm definitions in a loop over some parameter such as neighborhood size in order to plot accuracy as those parameters change.

We will not go into all the details here, but LensKit intercepts Groovy method calls to delegate evaluation directives to various constructors and `addFoo`/`setFoo` methods on LensKit classes. For example, to evaluate the `crossfold` block in listing 3.7, LensKit does the following:

1. Look up the `crossfold` method in a properties file on the classpath and find that it is implemented by the CrossfoldTask class.
2. Call the task class's constructor with the argument `"ml-100k"`.
3. Evaluate the block with a *delegate* (a Groovy mechanism for intercepting method and property references when evaluating closures or code blocks); this delegate implements the rest of the logic.
4. Translate the `source`, `holdout`, and `partitions` calls into calls to `setSource`, `setHoldout`, and `setPartitions` on the task class.
5. Call the `call()` method on CrossfoldTask to run the crossfold and obtain a list of train-test data source objects representing each of the 5 splits it will generate.

The resulting list is then handled by the delegate in use to configure the `trainTest` block; that delegate forwards the call of `dataset` with a list of data sets (returned from `crossfold`) into multiple calls to TrainTestTask's `addDataset` method.

Groovy's extensive functionality for customizing code evaluation allows us to provide an evaluation scripting syntax that reads like a structured, declarative configuration file, while allowing users to take advantage of a full programming language when their needs so require.

3.8.2 Data Sets

The evaluator operates with data primarily in two forms: delimited text files and binary rating files.

The primary data management tasks are as follows:

crossfold The crossfold task takes a single data source of ratings data and splits in into *N* partitions for cross-validation.

subsample The subsample task takes a single data set of ratings and randomly sub-samples them by user, item, or rating to produce a smaller data set.

pack The pack task takes a data set and packs it into a binary file for efficient access.

The default crossfold configuration splits the users evenly into *N* partitions. For each partition, the test set consists of selected ratings from each user in that partition, and the training set consists of those users' non-selected ratings along with all ratings from the users in the other partitions. User ratings can be selected by picking a fixed number of ratings (`holdout`), a fraction of the ratings (`holdoutFraction`), or picking all but a fixed number of ratings (`retain`). The test ratings can also be selected randomly or by timestamp (with later ratings going into the test set).

In addition to user partitioning, the crossfolder supports partitioning ratings evenly into *N* partitions, and creating *N* samples of fixed size of the users (allowing *N* partitions of

a large data set with fewer users per partition; this can decrease the time required to run experiments on data sets with large numbers of users).

Many algorithms benefit from having the rating data available in memory in order to train the model and compute predictions and recommendations. Repeatedly scanning a delimited text file is very time-intensive. Loading a large data set into the Java heap with an object per rating, however, takes a good deal of memory and places additional strain on the garbage collector. In early versions of LensKit, we tried to use the SQLite embedded DBMS to provide indexed access to ratings, but it did not perform nearly as well as in-memory data.

LensKit now uses packed binary rating files to provide efficient data access for recommender evaluation. These files are read using memory-mapped IO, so on systems with adequate RAM the data lives in memory and the operating system's cache manager can take care of paging data in and out of memory as appropriate. The bulk of the file consists of rating data in binary format, either (u, i, r) or (u, i, r, t) tuples. The tuples are stored in timestamp order. The file also contains user and item indices; for each user (resp. item), the index stores the user (resp. item) ID, its rating count, and indices into the tuple store for its ratings. The indices are stored in user/item order and can be searched with binary search. This format provides very memory-efficient storage and, when paired with fast iteration, time-efficient data access. In addition to the `pack` eval task, LensKit provides a `pack-ratings` command in on the command line (appendix A.8) to pack a rating file.

The data set tasks, along with additional helpers such as the `csvfile` builder to define a CSV data source, produce data set objects (either DataSource, for a single source of ratings data, or a TTDataSet for a train-test pair of data sources) that can be manipulated by the eval script or passed directly to other data processing classes or the `dataset` directive of the train-test evaluator. Data sets are identified by a name as well as optional attributes; attributes are stored in a map, and the output CSV files contain a column for each distinct

attribute name used across the data sets.

3.8.3 Measuring Performance

The evaluator takes a set of algorithms and a set of train-test data sets and evaluates each algorithm's accuracy over each train-test pair. For each algorithm and data set, the evaluator builds a recommender model (if applicable) over the training data and attempts to recommend or predict for each user in the test data. It runs various metrics on the recommendations or predictions and reports their results in a CSV file. Like data sets, algorithms can also have attributes associated with them that appear as columns in the CSV output; in this way, if there is a loop over values of some parameter, those values can appear in their own columns so the analysis code does not need to parse them out of algorithm identifier strings.

Metrics can report results in three ways: they can produce per-user data values, which will be included in an optional CSV file of per-user metrics; they can produce aggregate values over an entire experiment configuration (algorithm / data set pair); and they can write outputs to files entirely under their control. LensKit provides the following metrics:

MAE Mean absolute error of predicting test ratings, available in both user-averaged and global variants. This metric ignores unpredictable ratings.

RMSE Root mean squared error of predicting test ratings, available in both user-averaged and global variants. Like MAE, it ignores unpredictable ratings.

Coverage (predict) Measures the number of attempted and provided rating predictions to compute coverage.

Predict nDCG Normalized discounted cumulative gain [JK02] used as a rank accuracy metric over predictions. We rank the test items by predicted rating and compute the nDCG of this ordering, using the user's actual rating of each item as its utility or gain.

Predict half-life utility nDCG computed using a half-life decay for the discount function [BHK98]. This metric has the benefit of being rooted in a probabilistic model of user behavior, as well as discounting the second item (when using logarithmic discounting of base b, typically 2 in traditional nDCG, the first b items have maximum weight).

Top-N nDCG nDCG computed over fixed-length recommendation lists.

Top-N Precision and Recall Precision and recall computed over fixed-length recommendation lists.

The predict metrics use the algorithm's RatingPredictor to predict the user's ratings for the test items. The top-N metrics use the algorithm's ItemRecommender to produce a recommendation list. The candidate set, exclude set, and (if needed) set of 'good' items can all be configured; a common configuration uses an exclude set of the user's training items and a candidate set of either all items or the user's test items plus a random set of 'bad' items. We plan to add more metrics in the future.

Recommender evaluation is a subject of significant interest and research [GS09; Bel12]. The recommender systems research community is currently in the process of establishing best practices for robust and reproducible recommender research, particularly for offline experiments, where a diverse set of metrics and subtle variations in experimental protocols make research results difficult to reproduce or compare [KA13]. One of LensKit's aims is to reduce this confusion and provide a standardized evaluation platform [Eks+11], a goal

shared by the developers of other systems such as mrec[10] and RiVal[11]. As the research community develops consensus on best practices in experimental protocols and evaluation metrics, we will be adjusting LensKit to use those best practices by default (although perhaps not immediately, to provide a reasonable migration path for incompatible changes).

3.8.4 Improving Experimental Throughput

The train-test evaluator provides two important features for improving the throughput of recommender evaluation.

The first is support for parallelizing evaluations. On multicore systems, LensKit can run the evaluations for multiple algorithm configurations and/or data sets (within a single train-test task) in parallel. It can run the evaluations all together, or isolate them by data set (so that only one data set's structures need to be loaded into memory at a time — this is useful on low-memory systems or with very large data sets).

The second is the ability to identify and share common components between different algorithm configurations. For example, the neighborhood size does not affect the item-item similarity matrix in item-item CF; an experiment testing many neighborhood sizes will be faster and take less memory if it computes the similarity matrix once and using it for all experiments. LensKit automatically identifies the identical components of the configuration graphs of different algorithm configurations and arranges for such components to be computed once and shared. The caching logic uses Java's soft references to share the same in-memory representation of such components between all active algorithms that require them, while allowing them to be garbage collected when no longer needed. If a cache directory is configured, the common components will be written to disk so that they do not

[10] `https://github.com/Mendeley/mrec`

[11] `https://github.com/recommenders/rival/`

need to be entirely recomputed if they are garbage collected and then needed again. The evaluator uses the same logic as the web integration support (section 3.9) to identify shareable components.

With these two features, LensKit provides useful support for taking advantage of multicore shared-memory architectures for recommender evaluation.

3.9 Web Integration

Web server environments place particular requirements on the software that integrates with them. Typical Java web application servers, such as Tomcat, handle each HTTP request in a separate thread. When a request comes in, the request handler is started up; if it needs database access, it opens a connection (typically from a connection pool), does the required processing, and returns the database connection to the pool. Some architectures lease database connections to request handlers on an even shorter-term basis, such as once for each database operation[12].

With LensKit's use of dependency injection, all dependencies must be available before an object can be instantiated. For components that require database access, this means that the database connection must be available to create the required DAOs, after which the component itself can be instantiated.

A typical LensKit recommender algorithm will require both database access (to get the user's current ratings or interest profile) and model data (such as a factorized matrix) to produce recommendations. Rebuilding the model for each web request would be prohibitively expensive; we would prefer to compute the model once, load it into memory, and share the same model across all web requests. LensKit algorithms are designed for this: the model

[12]The Java drivers for MongoDB use this design.

object is stand-alone and thread-safe. It is built by a separate builder component, and a light-weight item scorer component combines the model and live data from the database to produce recommendations.

To implement and use this functionality, however, the software must do several things:

- Identify the components to be shared.
- Instantiate the shared components.
- Arrange for the shared components to be used to satisfy the dependencies of the per-request components.

It would certainly be possible to do this manually. However, that requires each algorithm developer to provide code to accomplish this separation for their algorithm (which may not work correctly for potential extensions of their algorithm), or for the application developer to build and maintain code to instantiate shared objects for the algorithm they are using (making it more cumbersome to change algorithms).

LensKit takes advantage of Grapht's support for analyzing and manipulating object graphs prior to instantiating them in order to provide implementation-independent support for these tasks (chapter 4 describes the Grapht side of these capabilities in much more detail). It takes a single description of the complete recommender component graph and identifies the *shareable* objects. Shareable objects can be pre-computed, shared between algorithm instances in multiple threads, and generally serialized to disk for use in other processes. It then instantiates the shareable objects and creates a new dependency injection graph with the pre-instantiated objects in place of their original configurations for use in later instantiations of the recommender. This is encapsulated in the RecommenderEngine type.

The workflow therefore looks like this:

1. Prepare a LenskitConfiguration describing the complete algorithm configuration. At this point, the developer does not need to consider at all what components will be shared and what ones will be reinstantiated.

2. Build a RecommenderEngine from the configuration, instantiating all shared components.

3. For each web request, ask the recommender engine to create a Recommender, encapsulating a fresh recommender combining the model with whatever database connections and other ephemeral resources are needed.

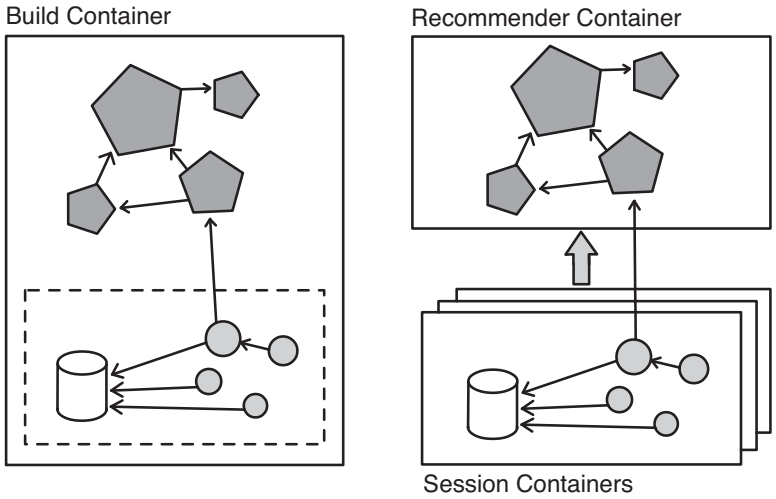


Figure 3.4: Object containers in a LensKit web application[13].

Figure 3.4 shows this in practice. Each configured object graph is encapsulated in a *container* (Recommender is a container, as is RecommenderEngine). The per-request recommender containers reuse objects from the shared container in the engine, in addition to the objects that must be isolated per request.

[13]Diagram by Michael Ludwig, published in [Eks+11].

To designate a component for pre-instantiation and sharing, the algorithm developer annotates it with the @Shareable annotation. Components with this annotation must be thread-safe and should generally be Serializable. LensKit will pre-instantiate and reuse such a component if and only if all of its dependencies are also shareable. This analysis means that if a shareable component is configured so that one of its dependencies that is generally shareable no longer is, it will automatically be downgraded to a non-shared component without the developer needing to do any checking or enforcement.

LensKit also provides a @Transient annotation for dependencies to indicate that a particular dependency should not be considered when determining a components shareability. If a component marks one of its dependencies as transient, it is promising that the dependency will only be used to build the object, and the built object will not retain a reference to it. For example, the item-item model builder's dependency on the data source is marked as transient, since it uses the data source to build the model but the final model is independent of it.

The final result of these manipulations is that each web request instantiates a set of lightweight objects that combine the current connection with heavyweight recommender components to provide the recommendation services of the rest of the application. We have successfully integrated this architecture with multiple web applications that are currently used in production.

3.10 Comparison with Other Systems

There are many other recommendation toolkits available, commercial, freeware, and open-source; section 2.4 listed some of them. Several of the open-source offerings now seem to be inactive (COFI, jCOLIBRI), some are focused on particular recommendation techniques

Feature	LensKit	Apache Mahout	MyMediaLite
Platform	Java	Java	C#/.NET
User-user CF	Yes	Yes	Yes
Item-item CF	Yes	Yes	Yes
Matrix factorization CF	FunkSVD	Yes	Many
Distributed algorithms	No	Yes	No
Visualization of configurations	Yes	No	No
Algo-independent lifecycle separation	Yes	No	No
Rating data support	Yes	Yes	Yes
Implicit feedback support	Partial[a]	Yes	Yes
Distinct data normalizations	Yes	No	No
Offline evaluation	Yes	Yes	Yes
Reuses shared components in eval	Yes	No[b]	No

[a] Finishing this is a high-priority project.
[b] Common component reuse may be achievable manually.

Table 3.1: Comparison of recommender toolkits

(myCBR), and others are focused more on providing recommendation services in applications (EasyREC, PredictionIO) than on supporting research and cutting-edge recommender system development or on particular integrations (e.g. RecDB [SAM13], providing recommender services within a database).

LensKit's most direct competitors are Apache Mahout and MyMediaLite. Apache Mahout is a machine learning library with support for many different algorithms, including several recommendation algorithms; it has extensive support for distributed computing [SBM12; Sch+13]. MyMediaLite [Gan+11] is a recommendation toolkit for the .NET platform (with good support for non-Windows systems via Mono) that has a particular focus on providing state-of-the-art rating prediction and item recommendation algorithms.

LensKit sets itself apart with its extensive support for research activities and its support infrastructure for connecting algorithms to evaluators and running applications. While we are playing catch-up in some areas, particularly advanced matrix factorization algorithms

and implicit feedback support, the algorithms and evaluations LensKit does are significantly more flexible.

As discussed in section 3.7, LensKit algorithms are built from many discrete pieces that can be replaced and recombined. This allows for extensive experimentation with distinct choices for similarity functions, data normalization methods, neighbor selection algorithms, etc., with very few limits on how they can be combined. Apache Mahout provides some configurability of its algorithms — the item similarity function can be replaced, for instance — but has relatively few configuration points; as near as we can tell, data normalization needs to be built in to either the data model (so the algorithm sees normalized data) or into each algorithm component itself. MyMediaLite supports reconfiguring algorithms via subclassing.

LensKit's evaluator is more flexible than either Mahout's or MyMediaLite's. Both Mahout and MyMediaLite support measuring an algorithm's performance on prediction accuracy or top-*N* metrics, but provide either a command line or a Java programmatic interface. With Mahout, the programmer must provide recommender builders that build testable recommenders. LensKit's ability to represent and analyze algorithms as entities allow it to train and evaluate algorithms using the same mechanisms used to load algorithm models for running applications, and basic evaluations read in a declarative fashion (evaluate of X algorithms, Y data sets, with Z metrics). LensKit's evaluator will also analyze the tested configurations to automatically determine components that can be trained once and shared between multiple configurations, providing a dramatic decrease in the cost of operations such as finding the best neighborhood size without requiring any additional effort from the programmer or researcher. LensKit also provides minimal entry points for new evaluation components such as metrics, and can run arbitrarily many metrics in a single evaluation pass; Mahout provides base classes to simplify writing new metrics, but the class embody-

ing a metric (or suite of metrics) drives the evaluation.

LensKit also has advantages for building applications around the recommender. Its support for separating pre-built and runtime components mean that the recommender integrator does not need to worry about what components can be precomputed and shared between requests, and what components do not (unless they need to debug a configuration that is not precomputing enough data): given an algorithm configuration, LensKit can instantiate the pre-computable portion, save it to disk, and instantiate the needed runtime components. All of this is in a configuration-independent fashion, so the recommender for an application can be changed simply by replacing its algorithm configuration file.

A key enabler of LensKit's lifecycle separation — as well as some of its evaluation optimizations — is that it treats algorithm specifications as objects that can be manipulated and analyzed. It can perform operations on a recommender algorithm or configuration itself, not just the models and components that comprise it.

Finally, LensKit is built from a somewhat different philosophy. As we see it, MyMediaLite and Mahout's APIs are structured around the idea that 'here is a recommender algorithm, connect it to your data and use it', with some options for configuration. LensKit is structured around a large collection of pieces that can be wired together to make a recommender, and a set of defaults and example configurations to put them together into common types of recommenders. In addition to affecting the design of algorithms, this also manifests in the public API: different recommendation services are provided by different components, and not all configurations will necessarily provide all services.

Both philosophies have advantages and disadvantages. It is currently easier to take Mahout or MyMediaLite off the shelf and get recommendations from it than it currently is with LensKit, but once LensKit is running it provides more built-in reconfigurability and flexibility in its algorithm components. It is possible to adapt Mahout or MyMediaLite

to a variety of configurations for research and experimentation, but LensKit provides the implementations and configuration infrastructure necessary to test many different variants out-of-the-box. We are, however, working on solving the getting-started problem through improved documentation, more example code, and simplified wrapper APIs.

3.11 Usage and Impact

Since we originally published LensKit in 2011 [Eks+11], it has seen use in several research projects. In our own research, we have used LensKit for the algorithmic analyses in the rating interface experiments we have run [Klu+12; Ngu+13], as well as the research described in the remainder of this thesis. Google Scholar records a total of 27 citations of the core LensKit paper [Eks+11][14]. LensKit also provides the recommendations for several live systems:

- MovieLens, operated by GroupLens Research, provides movie recommendation and tagging services. URL: `http://www.movielens.org`
- BookLens, also operated by Grouplens, provides book recommendations integrated with library card catalogs. URL: `http://booklens.umn.edu`
- Confer, from MIT CSAIL, is an online conference program site that uses LensKit to recommend papers for conference attendees to see and other attendees that they may wish to meet. URL: `http://confer.csail.mit.edu/`

From time to time, someone will post on the LensKit mailing list with a question about using LensKit in some new environment, and there are also likely other uses that we have not heard about.

[14] `http://scholar.google.com/scholar?oi=bibs&hl=en&cites=14771795286610726161`

Our MOOC on recommender systems (*Introduction to Recommender Systems* on Coursera) used LensKit as the basis for its programming assignments; we had 800–1000 users complete the programming assignments.

LensKit is also regularly brought up in discussions about reproducible recommender systems research. We encouraged the recommender research community to adopt a culture of publishing code built and tested against accepted, publicly-available recommender platforms to support new recommender algorithm and evaluation research. While this has not yet been established as a general norm, there is increasing interest in reproducible research and best practices for comparable recommender research, which is an encouraging sign. It will take the community time to establish best practices for evaluating recommender research, and the conversation seems to be going in profitable directions.

Chapter 4

Supporting Modular Algorithms

Many large software systems — LensKit included — comprise many components that work together to provide the system's functionality. Individual components seldom operate alone; many of them depend on other components to fulfill their responsibilities. In the last twenty years, the *dependency injection* (DI) pattern has seen wide adoption as a means of fitting together the components of such systems, and we have adopted it for designing and configuring LensKit's recommender implementations as described in section 3.7. Unfortunately, LensKit's needs are not well-served by existing dependency injection toolkits. The first versions of LensKit used Google Guice for instantiating algorithms; when it proved inadequate, we tried using PicoContainer, which was also a poor fit. We finally wrote Grapht, a dependency injection toolkit for Java, to manage LensKit's dependency injection with a new set of configuration and graph processing capabilities.[1]

Our work on dependency injection has been driven by two specific shortcomings with other toolkits with respect to LensKit's needs:

- Limitations on configuration that severely hinder the composeability of components. Most toolkits to not deal well with the same class or interface appearing in many places in the object graph with different implementations or configurations, and the tools they do provide for such scenarios are weak. This means components cannot be

[1] Michael Ludwig contributed significantly to the work in this chapter, particularly refining the design and writing the initial implementation. We are preparing this work to submit for publication.

freely reused and composed, but need to be wrapped in extra classes that know about how they fit into the final object graph.

- Inability to construct and process the object graph as a first-class object prior to instantiation made it difficult — if not impossible — to provide robust support for detecting & reusing common components in experiments, separating prebuilt and run-time components, and providing diagnostic and debugging support for algorithm configurations.

Grapht addresses both of these concerns: the first through *context-sensitive policy*, allowing objects to be configured based on where they are used, and the second by decoupling dependency resolution from object instantiation and exposing the resolved object dependency graph as an object that can be analyzed and manipulated. Grapht's internal architecture is built on this object graph abstraction and many of its features are implemented in terms of graph transformations; this has the side effect of making it amenable to formal treatment. We use a formal model of dependency injection — a abstraction of Grapht's design — to describe the key algorithms and to show that certain commonly-used dependency injection features are technically superfluous, replacable with strictly more expresive alternatives.

4.1 Dependency Injection

Dependency injection [Fow04; YTM08] is a design pattern arising from applying Inversion of Control to the problem of instantiating objects that have dependencies on other objects. If a component A requires another component B in order to fulfill its obligations, there are several ways that it can obtain a suitable reference. A can instantiate B directly (listing 4.1(a)); this is straightforward, but makes it difficult to substitute alternative implementations of

```
public UserUserCF() {
    similarity = new CosineSimilarity();
}
```

(a) Direct instantiation

```
public UserUserCF(SimilarityFunction sim) {
    similarity = sim;
}
```

(b) Dependency injection

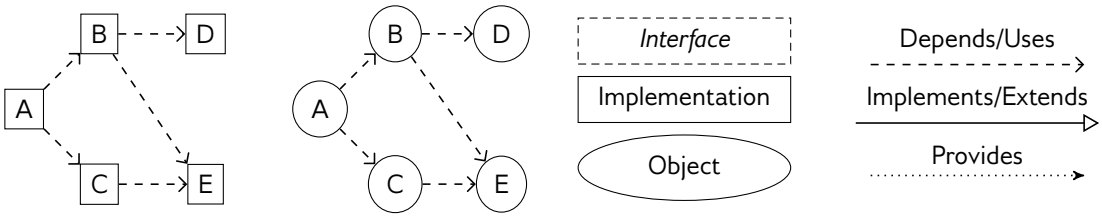
Listing 4.1: Constructors depending on another component.

B. A can also obtain B from some other service, like a factory or service locator, allowing alternative implementations to be used but making A dependent on the resolution strategy. Finally, in dependency injection, A can require B to be provided via a constructor argument (listing 4.1(b)). That is, the dependency (B) is *injected* into A. Whatever component creates A is therefore free to substitute an alternate implementation or reconfigure B in any way it wishes.

When used throughout the design of a system, dependency injection (DI) provides a number of advantages. Most follow from reduced coupling between components. Some of these advantages include:

- Components are free of all knowledge of the implementations of their dependencies — they do not even know what classes implement them or how to instantiate them, only that they will be provided with a component implementing the interface they require.

- Components can be reconfigured by changing the implementations of their dependencies without any modification to the components themselves.



(a) Class diagram (b) Object diagram (c) Shape legend.

Figure 4.1: Class dependencies, object graph, and legend.

- Components can be more easily tested by substituting mock implementations of their dependencies. While mocking is not new, the component design encouraged by dependency injection makes it particularly easy to substitute mock objects.

- Each component's dependencies are explicit, appearing as formal arguments of the constructor setup and initialization methods, so the component's interaction with the rest of the system is largely self-documenting. This can make the system easier to understand and more amenable to static analysis.

The reduced coupling and increased flexibility of dependency injection comes with a cost: in order to instantiate a component, its dependencies must be instantiated first and provided via a myriad of constructor parameters. This requires the code initializing a component to know its dependencies, construct them in the proper order, and pass them in to the constructor. Doing this manually, while possible, is cumbersome.

To make it easier to configure and instantiate software built around dependency injection, a number of toolkits called *dependency injectors* or *DI containers* have been developed. These toolkits take care of resolving the dependencies of the various components in a system, instantiating them in the proper order, and wiring them together. This automated support for dependency injection is sometimes called *autowiring*.

An autowiring DI container's task is to instantiate and connect a graph of objects that

will realize the application's functionality. This often takes the form of instantiating some root component along with its dependencies. Figure 4.1 shows an example; the DI container is responsible for transforming class A along with its dependencies ((a)) into a graph of objects ((b)).

To accomplish this task, the container must typically do three things:

1. Identify the dependencies of each component.
2. Find an appropriate implementation for each dependency.
3. Instantiate the final object graph, providing each component to its dependencies.

These tasks can be performed together, identifying and resolving dependencies lazily in response to object instantiation requests, or with a phased approach in which a dependency solution or instantiation plan is computed as an object in its own right and passed to a separate component to perform instantiation.

The DI container typically extracts dependencies directly from component class definitions using reflection or static code analysis. It generally uses reflection or code generation (either at run time or compile time) to instantiate the components. Both of these capabilities are dependent on the capabilities of the language and environment that the DI container targets; we discuss how Grapht performs these tasks in more detail in section 4.4.

4.2 Related Work

Dependency injection has seen significant use for at least ten years, with numerous toolkits providing automated support for it. In this section, we survey research literature and existing software packages relating to dependency injection.

4.2.1 Prior Research

Despite widespread use of dependency injection by the software development community, we have been able to find little treatment of the subject in the research literature. Yang, Tempero, and Melton [YTM08] empirically studied the prevalence of its use, and Razina and Janzen [RJ07] studied its impact on maintainability measures such as coupling and cohesion. DI has also been shown to be effective for configuring game components [Pas+10] and connected with aspect-oriented programming [CI05], but there does not seem to be much other published research on dependency injection since Fowler [Fow04] provided its modern formulation. In particular, there has been little treatment of the core principles of dependency injection, the facilities needed for effectively supporting it, and effective models for reasoning about its capabilities, limitations, and potential extensions.

Some work on component instantiation anticipated aspects of dependency injection. Magee et al. [Mag+95] describe the Darwin notation for describing software component wiring; it is formalized in terms of the π-calculus, and seems amenable to static analysis, but required the entire system component graph to be explicitly specified.

4.2.2 Existing DI Containers[2]

There are many dependency injection containers available for Java and the .NET platform. These runtimes support reflection and generally target statically typed languages, making type-safe runtime dependency injection easier. JSR 330 standardizes dependency annotations and behavior for DI containers in Java, and most Java DI containers are adding JSR 330 support if they do not have it already. JSR 330 itself is based heavily on the design of Google's Guice DI container; Spring and PicoContainer are also used significantly in Java applications and have implemented JSR 330.

[2]Michael Ludwig conducted the survey of existing implementations.

Spring is an expansive application framework, providing tools for web development, aspect-oriented programming, and dependency injection, among other things. Early versions of Spring used XML descriptions of the complete object graph for dependency policy. Although verbose, this is a powerful way of configuring dependencies and can achieve the same results context-sensitive configuration. Spring has more recently been updated to support the JSR 330 annotations and automatically configure certain types. Although IDE support for statically analyzing Spring configurations exist, the dependency solution graph is not a first-class entity in Spring's IoC framework, making it difficult to do static analysis of object configurations.

Guice and PicoContainer function similarly from a high level. Both provide a Java API to specify dependency configuration and lazily resolve dependencies while instantiating objects (with the consequence that there is no pre-computed dependency plan). They provide hooks that can intercept and record the dependency solution as its being discovered, and tools exist that use this to produce static dependency graphs. Since this instrumentation is limited to operating while objects are being instantiated, however, it does not allow the solution graph to be statically computed and analyzed before instantiating objects. Guice also supports extensive defaulting (called *just-in-time* injection), looking up default implementations from Java annotations; PicoContainer requires all component implementations that will participate in injection to be explicitly described.

Guice, Spring, and PicoContainer all provide support for integrating with web frameworks in various ways, so that the framework uses the DI container to instantiate the request handlers and other objects. Related to this is the support for *scopes* and annotations to specify them, controlling whether an object is instantiated for each request, session, or shared over the server's lifetime. All 3 provide support for scopes, although PicoContainer implements them differently. Finally, some containers also provide support for lifecycle manage-

ment, starting and stopping objects to release external resources in addition to instantiating them.

Outside the Java ecosystem, Ninject [Koh12] provides similar DI services for .NET. Ninject supports both just-in-time binding and context-sensitive binding with an elegant API that avoids the verbosity of Spring's context-sensitive solutions. In Ninject, bindings are provided the injection context and can invoke an arbitrary boolean function to determine if it matches. When instantiating or "activating" objects, Ninject proceeds in a lazy fashion like Guice, allocating new instances as necessary to satisfy the parameters of a required instance. As far as we know, Ninject does not provide any support for static analysis or manipulation of object graphs.

Table 4.1 summarizes the Java container implementations we have discussed. Although each of these is fully capable and useful as a general DI container, none of the examined implementations cleanly supported the three key features we desired for meeting our goals for LensKit. We do note that Grapht does not completely cover their functionality; to date, we have focused on developing the capabilities that LensKit requires. It is certainly feasible to implement many of these features on top of Grapht, but it has not yet been a priority.

4.3 Requirements

Grapht's requirements are derived from JSR 330, the common specification for Java dependency injectors; the behavior of other DI containers; and the needs of LensKit. We are not interested in needlessly inventing new APIs, and have been pleased overall with the programmatic interface and behavior of Guice's basic functionality. We adopt its paradigm and terminology for configuring the DI container, simplifying and extending it to meet LensKit's particular requirements. This should aid the transition for developers already familiar with

	Guice	Spring	PicoContainer	Grapht
Configuration	Java	XML	Java	Java[a]
Static Analysis	✗	✗	✗	✓
Context-sensitive Policy	✗	Hard	✗	✓
Just-in-Time Binding	✓	✓	✗	✓
Scope Annotations	✓	✓		✗
Web Framework Integration	✓	✓	✓	✗[b]
Lifecycle Support	✗	✓	✓	✗[c]

[a]LensKit provides additional support for Groovy-based config files.

[b]It is easy to integrate Grapht with some frameworks, but we do not provide any out-of-box support. The Grapht wiki does provide a snippet for integrating with Play.

[c]We plan to add lifecycle support to Grapht before releasing LensKit 3.0.

Table 4.1: Summary of DI Containers

Guice (or another toolkit), as much of their knowledge will transfer. Many of the capabilities described in this section are also standard behavior for DI containers; context-sensitive policy and first-class object dependency graphs are the new requirements that we impose.

4.3.1 Basic Policy

In order to instantiate an object graph, the DI container must know what classes should be instantiated to satisfy each dependency. For example, to instantiate a LensKit recommender, it needs to know what implementation of ItemScorer should be used.

If all the dependencies in question are concrete classes (as in Fig. 4.1(a)) or the runtime environment has no concept of interfaces or polymorphism, then resolving dependencies is simply a matter of looking up the constructor for each dependency.

If there are multiple implementations of component interfaces to choose from, the DI container needs some form of *dependency policy* [Mar96] to determine which implementation to use to satisfy each dependency. Figure 4.2 shows a simple class diagram where a dependency policy is necessary; interface I has two implementations C1 and C2. When

(a) Class diagram

(b) Potential object graphs

Figure 4.2: Interface with multiple implementations.

instantiating A, the injector needs to know which implementation to use to satisfy the dependency on I; both graphs in Fig. 4.2(b) are valid solutions, and which one is desired may depend on many factors.

Policy can be provided by fully specifying the object graph, e.g. in an XML file. It can also be defined by *bindings* from component types (typically interfaces or abstract classes) to concrete classes implementing those types; this is the approach used by Guice and Grapht. To produce the top graph in fig. 4.2(b) from fig. 4.2(a), the binding $\mathsf{I} \mapsto \mathsf{C1}$ would be used; the binding $\mathsf{I} \mapsto \mathsf{C2}$ produces the bottom graph.

It is also useful to be able to bind interfaces directly to pre-instantiated objects or to *providers*. A provider is a component with a single method, `get()`, that returns the object to be injected; when provider bindings are supported, the provider can either be specified by class (in which case the injector resolves the provider's dependencies and instantiates it like it would any other component) or by binding to a provider object. A DI container can also support a mix of binding-based and specified-graph policy, allowing an interface to be bound to a specified subgraph, which may in turn have unresolved components that must be resolved using binding-based policy.

One final source of dependency policy is *just-in-time* dependency resolution. With just-in-time resolution, concrete classes are injected without requiring configuration and interfaces can carry annotations specifying their default implementation. Supporting just-in-time resolution is important to enable the extensive automation and defaults that we desire

(a) Class diagram (b) Bindings (c) Example object graph

Figure 4.3: Component with qualified dependencies (indicated by edge labels).

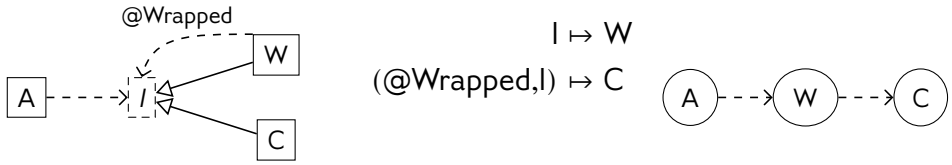
from the DI container; many of LensKit's interfaces have annotations specifying default implementations to minimize the configuration work required to get a working recommender.

Explicit bindings override just-in-time resolution decisions. Bindings and defaults complement each other well, and together provide a good deal of flexibility for refactoring and revising components. Classes can be redesigned to introduce new intermediary components, or to change what class uses some dependency; so long as the key reconfigurable interfaces remain, these changes do not have to break application configurations.

4.3.2 Qualifiers

Type information is not always sufficient to describe a component's dependencies. There are cases, such as that shown in Fig. 4.3, where the same component interface is used in different roles and the desired configuration will use different implementations in each of these roles. To accommodate this, JSR 330 defines *qualifiers*, annotations added to injection points or dependency declarations to provide additional information to the dependency resolver and allow the policy to specify different implementations for the same interface in different settings[3]. In Fig. 4.3(c), component type B is bound to different implementations in the left and right positions.

[3]Similar concepts are applicable in other environments as well; any mechanism for associating additional metadata with a component's dependencies can be used to implement qualifiers.



(a) Class diagram (b) Bindings (c) Object graph

Figure 4.4: A wrapper component. W implements I by wrapping another component of type I, in this case C.

Another use of qualifiers is to enable wrapper components to be created. As shown in fig. 4.4, these components implement an interface by wrapping another component of the same interface. The wrapper annotates its dependency on the wrapped component with a qualifier so the policy can distinguish between the primary binding (interface to wrapper) and the wrapped binding (qualified qualified interface to implementation). Wrappers are used extensively in LensKit to allow data pre- and post-processing to be decoupled from more fundamental computation.

In LensKit, these two needs combine when we want to configure a hybrid component. Figure 4.5(a) shows the constructor for a hybrid item scorer that computes a linear blend of two other scorers ($s(u,i) = ws_{\text{left}}(u,i) + (1-w)s_{\text{right}}(u,i)$). Qualifiers allow the left and right scorers to be configured differently, and to be configured separately from the main item scorer. Figure 4.5(b) shows bindings that will configure the blending item scorer to be the primary item scorer, and set it up to blend the results of user-user and item-item CF with equal weight.

4.3.3 Context-Sensitive Policy

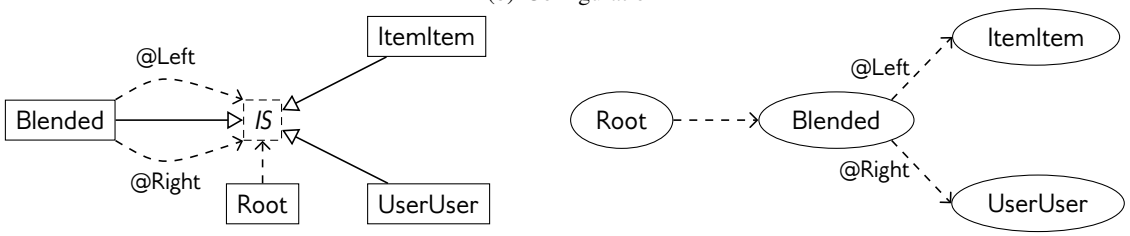
While qualifiers allow dependency policy to be conditional on annotations indicating how a dependency is going to be used, they have limitations when applied in larger object graphs. In Fig. 4.6(a), A has two qualified dependencies on I1. Unlike the case in Fig. 4.3 where we

```
public class BlendingItemScorer implements ItemScorer {
    @Inject
    public BlendingItemScorer(@Left ItemScorer left,
                              @Right ItemScorer right,
                              @BlendWeight double weight) {
    }
    /* ... */
}
```

(a) Constructor with dependencies

```
bind ItemScorer to BlendingItemScorer
bind (@Left,ItemScorer) to ItemItemScorer
bind (@Right,ItemScorer) to UserUserItemScorer
bind (@BlendWeight,Double) to 0.5
```

(b) Configuration

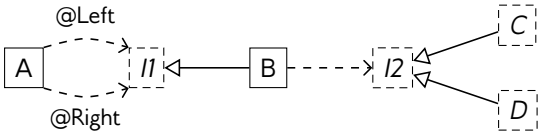


(c) Class and object diagrams

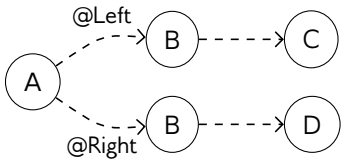
Figure 4.5: Hybrid item scorer.

want to use different implementations of I1 for the left and right components, in this example we want to use the same implementation with different configurations for its dependency on I2. In LensKit, this would arise if we wanted to adapt fig. 4.5 to blend two ItemItemScorers with different configurations instead of two different item scorer implementations. A similar problem arises, perhaps more naturally, if we want to use the baseline subtracting normalizer in two places subtracting different baselines.

Context-sensitive bindings allow these kinds of graphs to be configured. A context-sensitive binding is only activated in certain portions of the object graph: in this case, the bindings for I2 depend on whether they are being used to satisfy some (transitive) depen-



(a) Class diagram



(b) Desired object graph

$\vdash$ I1 $\mapsto$ B
(@Left,B) $\vdash$ I2 $\mapsto$ C
(@Right,B) $\vdash$ I2 $\mapsto$ D

(c) Bindings

Figure 4.6: A dependency graph requiring context-sensitive policy. $X \vdash I \mapsto C$ denotes that I is bound to C only when satisfying dependencies of X.

dency of the left or the right B component. These bindings depend on the *context* of the dependency, a path through the dependency graph from the initially-requested component to the component whose dependency is to be satisfied.

The fundamental problem solved by context-sensitive policy is that dependency solutions guided only by type- and qualifier-based policy lack composability. B can be configured in two different ways, but qualifiers do not have sufficient expressiveness to allow those two different configurations to be used as subgraphs of a larger object graph.[4] Allowing bindings to depend on context increases their expressiveness and allows more complex object graphs to be described by bindings without needing to resort to providers, pre-instantiated instances, or explicit descriptions of subgraphs.

We show in section 4.6.4 that qualifiers can be replaced with additional types and coupled with context-sensitive policy to achieve the same results as qualifier-based policy. Thus, context-sensitivity is a more general solution to the same set of problems as those

[4]Technically, any graph can be configured by binding to pre-instantiated objects, providers, or specified subgraphs. We find such graph specification to be cumbersome, and unnecessary use of instances and providers can hinder static analysis.

solved by qualifiers. We continue to use qualifiers because they are specified by JSR 330 and because they are syntactically convenient.

In LensKit, we generally use qualifiers to differentiate multiple dependencies on the same type in a single component (as in the hybrid situation), and with primitive types to define parameters such as neighborhood sizes. We occasionally use them to identify a particular role that a component plays, such as the @UserSimialrityThreshold qualifier applied to some dependencies on Threshold to indicate that the threshold will be used to threshold user similarities, making it easy to configure a single threshold to be used everywhere user similarities are thresholded.

4.3.4 Subtype Binding

Components are not limited to implementing a single interface. Interfaces can extend and refine other interfaces, and components can implement multiple distinct interfaces. In LensKit, this is frequently the case for data access objects: the JDBC-backed DAO, for instance, implements all of the standard DAO interfaces (section 3.5) in a single object. We also encourage application code to extend our DAOs — while this is not necessary, it is common for an application that exposes additional information about items, for example, to do so by extending ItemDAO with methods like `getItemTags(long)`.

Naïvely, using the JDBC DAO for all data access would require a separate binding for each DAO interface. We want the DI container to have reasonable behavior with respect to the subtypes and supertypes of the interface and class explicitly involved in a binding. By 'reasonable', we mean that we want the behavior to provide useful capabilities or convenience while adhering to the Principle of Least Surprise (a subjective standard, but helpful as a guiding principle). We do not want spurious and confusing bindings to arise from straightforward policy statements.

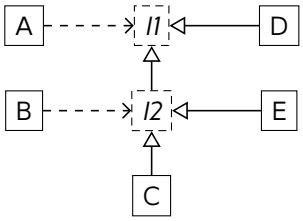


Figure 4.7: Class graph with subtypes

Consider the class diagram in fig. 4.7, where there are two interfaces and A depends on I1, while B depends on the more specific I2. For convenience, we would like the single binding I1 ↦ C, or even I2 ↦ C, to satisfy both dependencies. Requiring both bindings to be explicitly specified would increase both maintenance burden and the difficulty of figuring out exactly what binding rules are needed when building a configuration. However, either decision should be overrideable; if there explicit bindings I1 ↦ C and I2 ↦ E, C should not be used to satisfy B's dependency. Also, a binding I1 ↦ D should clearly not satisfy B's dependency, as B requires a I2.

The particular policy we have devised is as follows:

1. Explicit bindings always take priority.

2. The binding A ↦ B should be treated as if it also bound every type that is a supertype of A, or a supertype of B and subtype of A, to B, unless such a binding (called a *generated* binding) conflicts with an explicit binding.

In the case of the JDBC DAO, we simply bind the DAO type to itself, producing generated bindings for all of its interfaces.

4.3.5 First-Class Object Dependency Graphs

To implement the various features that arise from LensKit being able to treat algorithms as first-class, manipulable objects — automatically identifying shareable components for reuse or prebuilding, diagramming configurations, etc. — we need the dependency injector to provide access to the object graph. This should ideally happen without instantiating any objects, as some objects are very expensive to instantiate.

This can be achieved by separating dependency resolution from object instantiation. If the dependency injector first resolves the dependencies, building up a graph of the concrete classes, providers, and instances that it will use to assemble the final object graph, and then instantiates objects according to this plan, then the plan can be analyzed and modified in the middle.

4.3.6 Non-Requirement: Circular Dependencies

Many DI containers support circular dependencies among components, and JSR 330 mandates this capability. Grapht has optional support for circular dependencies in order to comply with the JSR and pass its compatibility tests, but LensKit disables this support.

There are a variety of reasons that it is good to avoid circular dependencies:

- Instantiating circular dependencies is awkward, as objects need to be able to be partially initialized and passed to each other before initialization is complete.
- Because objects can obtain references to partially-initialized objects during the instantiation process, there is greater opportunity to misuse objects (invoke methods on them before initialization is complete). In the absence of circular dependencies, a reference to an object is only made available to other components once the object is

fully instantiated and initialized, so it is impossible for this particular type of runtime error to occur.

- Circular dependencies can often be factored out, resulting in more loosely-coupled class design. This can be done by either injecting both components into a third that mediates their interaction, or injecting a common component into each of the formerly mutually-dependent components; this would be done, for instance, when refactoring components built on the observer pattern to use a publish-subscribe event bus instead.

In Java, and many other environments, circularly dependent components cannot be instantiated if their dependencies are only expressed via constructor parameters. The cycle must be broken either by having some dependencies injected into fields or setter methods, or by injecting providers of required components rather than the components themselves at some point in the cycle. In both cases, the components must allow for the circular dependency in their design, either by depending on a provider or by exposing dependencies via fields or setters.

4.4 Implementation of Grapht

Grapht is our open-source Java dependency injector. It is compatible with JSR 330 [JL09] and passes its TCK, so it can be used as a replacement for existing containers in many situations. The code base is less than 5000 lines of Java, excluding tests; its multi-phase design and graph-based approach have enabled us to build a simple, clean implementation that provides the features we require and many of the features generally expected of dependency injectors. There are a number of features provided by other systems such as Guice that we do not yet provide, mostly because we have not yet needed them, but most of them should not be difficult to implement on top of Grapht's architecture if they are ever needed.

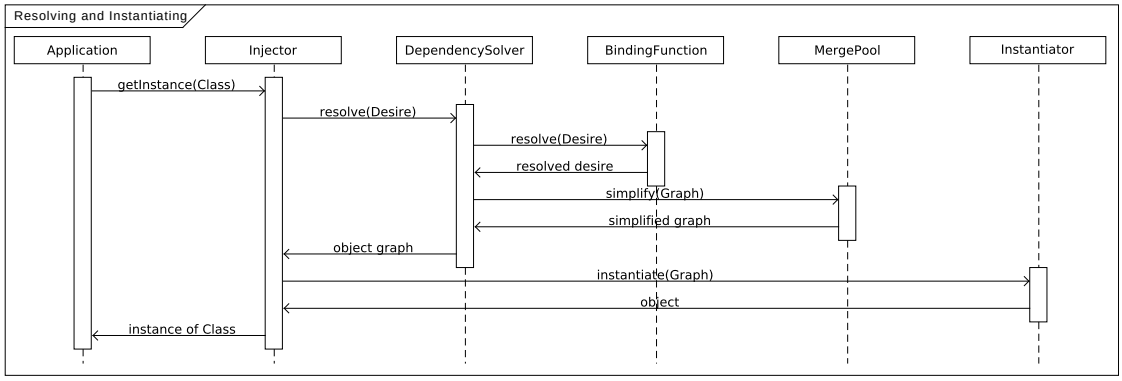


Figure 4.8: Simplified instantiation sequence diagram.

4.4.1 System Overview

Grapht's design revolves around constructing and using a graph of objects to instantiate. This graph represents the graph of objects that will result from the dependency injection operation.

Figure 4.8 shows the high-level components that work together to resolve and instantiate an object. The application (far left) requests an instance of a class, and the remaining components (all part of Grapht) work together to do the following:

1. Resolve the class to an implementation.
2. Recursively resolve its dependencies.
3. Simplify the graph to identify shared components.
4. Instantiate the object.

The following sections describe in more detail how these different components work.

4.4.2 Desires, Satisfactions, and Dependencies

Grapht uses a pair of abstractions for representing requests for components and the implementations that will be used to satisfy them. A Desire is an abstraction of a component request, generated by either the application or the dependencies of another component. It encapsulates the type of the desired component, its qualifier, and other attributes that may be associated with a dependency. For desires that arise as the dependencies of other components, the desire also keeps track of the constructor or setter parameter that generated it, primarily for debugging and diagnostic purposes.

Desires are resolved to Satisfaction objects, abstracting a means of instantiating or obtaining an object that will satisfy the desire for a component and the dependencies required to instantiate such an object. Satisfactions have two primary operations: they can report their dependencies (as a list of desires), and can create a component instantiator given a map from the satisfaction's dependency desires to instantiators of the required components.

Grapht implements several types of satisfactions, corresponding to the different types of binding policy targets discussed in section 4.3.1:

ClassSatisfaction instantiates a class, depending on the dependencies extracted from the class's injection points.

InstanceSatisfaction provides a pre-instantiated object and never has dependencies.

ProviderSatisfaction uses a pre-instantiated Provider object to produce instances, and never has dependencies.

ProviderClassSatisfaction instantiates a Provider class, extracting dependencies from the provider's injection points. It instantiates the class just like ClassSatisfaction, and

then invokes the resulting provider object's `get()` method to obtain the instance of the desired component.

NullSatisfaction is a special case of InstanceSatisfaction that returns `null` and contains extra book-keeping data needed to track the type of component that is not being returned.

A desire can have a satisfaction associated with it. Such a desire can be instantiated as-is without further resolution, although it may yet be resolved to a different satisfaction.

As Grapht is built to be JSR 330-compliant, the satisfaction implementations use the Java reflection API and the annotations defined by JSR 330 to identify injectable classes and their dependencies. It can inject dependencies into constructor parameters, setter methods, and fields; we prefer constructor and setter injection, but field injection is available for compatibility with other DI containers.

JSR 330 defines an annotation @Inject that is used to identify constructors, setters, and fields that participate in dependency injection. A class can be instantiated as a component if it has a no-argument constructor or a constructor annotated with @Inject. The arguments of that constructor, along with the arguments of all @Inject-annotated setters and fields, are taken as the component's dependencies. Qualifiers are also specified as Java annotations; any annotation that is itself annotated with @Qualifier (also defined by JSR 330) can be used as a qualifier.

Satisfactions that instantiate classes (ClassSatisfaction and ProviderClassSatisfaction) scan the constructors, methods, and fields of their classes they encapsulate to determine all the dependencies and encapsulate them in desires.

4.4.3 Resolving Desires and Dependencies

The dependency solver is responsible for taking a description of a root component, determining how to instantiate it, and recursively building an instantiation plan for each of its dependencies. The dependency resolution logic itself is separated from the messy details of Java reflection, keeping it relatively self-contained and clean.

At a high level, the dependency solver operates as follows:

1. Determine the satisfaction for the current desire.
2. Recursively build graphs for each of the satisfaction's dependencies.
3. Create a graph whose root is labeled with the current desire's satisfaction, with outgoing edges to the subgraphs of for the dependencies.

The dependency solver uses a list of binding functions (together representing the configuration policy) to resolve each desire to its appropriate satisfaction. The internals of these binding functions are described in sections 4.4.5 and 4.4.6, but they interface they expose to the dependency solver consists of a single method, `resolve`, that takes a desire and a context and returns the resolution of that desire. The resolution is itself another desire, which may be subject to further bindings.

The dependency solver scans through the list of binding functions, looking for one that can supply a resolution for the current desire. Once it has found a resolution, it scans again to see if any binding functions have a binding for the desire produced by the previous round. This is repeated until no binding functions produce new resolutions for the desire, or a binding function indicates that its result should not be subject to further re-binding.[5] The

[5]Bindings to pre-instantiated instances and to providers are not subject to re-binding. This decision may be revisited in the case of bindings to provider classes.

iterative resolution algorithm allows policy to specify specify that some class should be used to implement an interface, and then separately specify how that class should be instantiated.

If no resolution can be found for a desire, then the dependency solver raises an error unless the desire is optional. Optional desires arise from dependencies annotated with @Nullable; in this case, Grapht produces a solution that satisfies the desire with a null reference.

The list of binding functions typically consists of one or more rule-based binding functions, containing the bindings from the application-specified policy at different levels of precedence, followed by a binding function that looks up defaults.

The dependency solver maintains a *desire chain* when resolving any component request or dependency. The desire chain is the sequence of desires that have been encountered in resolving that component. In the initial pass through the binding functions, it will be a singleton list containing the initial desire; each subsequent pass will have another desire appended to the chain. Binding functions receive the entire chain, not just the current desire; this allows them to modify their behavior based on whether they are matching an initial desire or the result of a previous binding function, and the rule-based binding function uses the chain to ensure that no rule is applied twice in the same chain (to avoid infinite loops).

The dependency solver produces its graph in a two-step process. First, it resolves all dependencies directly, producing a tree of satisfactions: if the same component is used in two places, it produces two graphs. It then asks a MergePool to simplify the graph. The merge pool uses a dynamic programming algorithm (described in section 4.6.2) to identify identical component subgraphs and coalesce them, producing a graph that has exactly one vertex for each unique component configuration (a satisfaction applied to a unique set of dependencies). The final graph, therefore, represents each object that could possibly be shared between components requiring it as a coalesced subgraph. Whether or not the object is actually shared is determined by a caching policy, but the graphs produced by the

dependency solver capture all possible sharing.

4.4.4 Representing Graphs

Grapht represents object graphs (instantiation plans) as rooted DAGs whose vertices are labeled with Component objects. A component consists of a satisfaction and any additional configuration related to instantiating the component. Currently, that additional information consists of the caching policy, specifying how instances of that component are to be reused: either memoize, always reinstantiate, or use the default policy configured on the injector or instantiator. This allows a global object reuse policy to be configured but overridden on a component-by-component basis as needed.

The edges of the constructor DAG are labeled with Dependency objects. A dependency encapsulates the desire chain that was followed to resolve that dependency, along extra flags or configuration relevant to that dependency edge to support extra features. This extra information currently consists of flags that control Grapht's graph rewriting capabilities (not described here).

4.4.5 Representing Bindings

The primary binding function implementation used in Grapht is RuleBasedBindingFunction, a binding function that is based on a set of individual bindings (*bind rules*, in Grapht's internal terminology). A bind rule has two responsibilities:

- Determining whether or not it matches a desire.
- If it matches a desire, returning its target: a desire (often with an associated satisfaction) encapsulating the type or object to which the matched desire has been bound.

The rule-based binding function manages the bind rules along with the contexts in which they are active, identifying and selecting an appropriate rule (if any) when each component request comes in.

There are several complicated design decisions in determining how to select an appropriate bind rule. Our guiding principle in making these decisions is to attempt to define behavior that provides the least surprising results. This is a somewhat subjective standard, we admit. However, there is significant prior art in dependency injection that we can look to for guidance: we aim to have similar behavior as Guice unless there is a clear reason to diverge. We also draw from our own experience developing and using LensKit and other software built on dependency injection to identify the kinds of configurations that may need to be expressed to meet non-degenerate requirements and attempt to design the rules to allow such configurations to be expressed in a clear but concise fashion.

Rule Matching

Whether a bind rule matches is determined by the type and qualifier (if any) of the desire being matched, and the context in which the desire arises. The bind rule has a type (the type on the left hand side of the bindings in section 4.3.1) and a *qualifier matcher* to determine this match, and the binding function associates it with a *context expression* identifying the contexts in which it is active.

The bind rule matches if its type is the same as that of the desire (subtyping is handled by emitting multiple bind rules, discussed later) and the qualifier matcher matches the qualifier. The following qualifier matchers are defined:

- Any, matching any qualifier (including the absence of a qualifier).
- Null, matching only unqualified dependencies.

- CLASS(τ), matching any qualifier of type τ. Java annotations are objects and can have types, just as any other object.
- VALUE(q), matching only qualifiers equal to q.
- DEFAULT, matching $q_\perp$ and any qualifier whose defining annotation type indicates that it should be matched by default. This is allow applications to define qualifiers that can be targeted by specific bindings but, in the absence of such a binding, should fall back to whatever implementation would be used for that dependency if it were not qualified.

If a qualifier is not specified when a binding is configured via the API, the DEFAULT matcher is used. We do not use value matchers very much, as the qualifiers we define and use tend not to have any parameters (so all instances of a particular qualifier annotation are equal to each other). Some qualifiers do define parameters, however, such as the @Named qualifier provided by JSR 330 as a light-weight way to qualify dependencies with arbitrary names.

Contexts are represented as paths from the root of the object graph to the component whose dependency is being resolved by the current desire; each element of the path consists of a qualifier and a satisfaction. Context expressions are regular expressions that match contexts in which the associated bind rules should be considered.[6] Each atom in the expression consists of a qualifier matcher and a type; unlike the types on bind rules, the type in a context expression atom matches any satisfaction producing a component of that type, including subtypes.

[6]We do not yet implement the full semantics of regular expressions, as we have not yet needed either to support our primary API or any use cases we have encountered. A future version of Grapht will likely include full regular expression support for completeness.

Selecting a Rule

The rule-based binding function tests all bind rules whose context expressions match the current context. If only a single rule matches, then it returns the result of that rule.

If more than one rule matches, it must determine which rule to use. This is determined by a concept of specificity: the most specific binding is used.

Specificity is determined first by the binding's qualifier matcher. NULL and VALUE matchers are considered the most specific, followed by CLASS, followed by ANY, and lastly DEFAULT. ANY is more specific than DEFAULT since it must be explicitly applied, and therefore reflects a more specific intent on the part of the application developer.

If multiple bindings with the same qualifier matcher match the dependency, the binding function considers the specificity of their context matches. This specificity is determined as follows:

1. For each element in the context, construct a priority for the match of that element. Wildcard matches have the lowest priority, followed by negated matches. Matches of an atom (a type and qualifier matcher) are prioritized based on the qualifier matcher priority[7] followed by *type distance*, with lower type distances having higher priority. The type distance is the number of types between the component type produced by a satisfaction and the type specified by the context expression element. If the expression element matches exactly the component type of the satisfaction, the type distance is 0; if it is an immediate superclass, the distance is 1, and so on.

2. Order the context matches by reversing them and sorting the reversed matches in lexicographical order by priority.

[7] Grapht does not currently consider qualifier matcher priority. This omission will be corrected in an upcoming release.

These rules are designed to provide reasonable behavior when coupled with the policy API (section 4.4.8) by favoring context expressions that match deeper in the object graph. This allows the programmer to specify policy by matching some type deep in the graph, and know that their new binding will take precedence over bindings further out in the graph.

If two matching bindings in a single binding function are equivalent — they have the qualifier matchers of equal specificity, and their context matches are equal — then Grapht fails with an error indicating that the bindings are ambiguous. Applications should not specify multiple applicable bindings without using multiple binding functions to indicate a clear precedence relationship.

Subtypes

Section 4.3.4 describes the basic requirements for the behavior of bindings with respect to subtypes. It informs our answers to the following questions:

- How do bindings match subtypes or supertypes?
- If multiple bindings match, how do we pick the correct one?
- It seems that context expressions should match subtypes — how do we order context matches in the face of subtypes?

The last of these was addressed previously when discussing context specificity; we now turn to the other two questions. One way to obtain the desired behavior would be to adjust the binding match rules so that a binding of a type τ also matches dependencies on supertypes of τ and supertypes of the target type τ' (optionally restricted so that they must also be subtypes of τ). This has the downside, however, that determining whether a binding matches would involve looking both at its match rule — the type, qualifier matcher, and

context expression — and its target, complicating the bind rule matching logic and semantics. It is cleaner, in our opinion, to have the question of whether a bind rule matches or not be completely determined by its left-hand side.

Therefore, we offload the subtype binding requirements to the policy builder. Bind rules only match desires if their bound type is identical to that of the desire. The policy builder builds three binding functions, each containing bindings for a different piece of the type hierarchy. For a binding of type T to an implementation C, the following bindings are generated:

1. An *explicit* binding $\mathsf{T} \mapsto \mathsf{C}$.
2. An *intermediate* binding $\mathsf{U} \mapsto \mathsf{C}$ for every type U that is both a subtype (exclusive) of T and supertype (inclusive) of C.
3. A *supertype* binding $\mathsf{S} \mapsto \mathsf{C}$ for every type S that is a supertype of T.

These binding functions are listed in decreasing order of precedence: exact bindings take priority over intermediate bindings, which take priority over supertype bindings. This allows for extensive auto-binding based on a few explicit bindings, but also allows the application to override bindings at any point in the hierarchy to redirect some interface to a different implementation.

The full list of binding functions used by the default configuration of a dependency solver is as follows:

1. Explicit bind rules
2. Intermediate bind rules
3. Supertype bind rules

4. Provider bindings (optional — Grapht can allow components to depend on providers of their dependencies instead of the dependencies themselves, and this is implemented as a special binding function; see section 4.4.9)

5. Defaults (section 4.4.6)

4.4.6 Defaults

The DefaultDesireBindingFunction implements Grapht's default (just-in-time injection) logic, allowing interfaces to specify default implementations or providers of themselves. This binding function is generally configured last in the DependencySolver's binding function list, so it is only consulted if there are no explicit bindings matching a desire.

Defaults can be specified in two ways: with annotations, or with property files. Grapht provides the annotations @DefaultImplementation and @DefaultProvider. These annotations can be applied to a class or interface to specify the default implementation of that type; they are equivalent to binding the type to the specified implementation or provider class. They can also be applied to a qualifier; when resolving a qualified dependency for which there is no binding, Grapht will first consult the qualifier for defaults before examining the annotations on the dependency's type.

Grapht also looks on the Java classpath for Java property files. For a type `package.Iface`, it looks for the file `META-INF/grapht/defaults/package.Iface.properties`; if such a file exists, then it is expected to have an `implementation` or `provider` setting that specifies the name of an interface or provider class, respectively. This mechanism allows applications to specify default implementations for classes that they import from other, non-injection-aware packages.

4.4.7 Instantiating Objects

Once a component and all of its dependencies have been resolved and the final graph simplified, the component can be instantiated. The role of instantiator in fig. 4.8 is filled by two objects in Grapht. The InstantiatorFactory converts an object graph into a component instantiator (currently defined by reusing the Provider interface). It uses the root node's Satisfaction to obtain the component instantiator, resolving its dependencies using the outgoing edges. This is a recursive process, using each subsequent node's outgoing edges to provide instantiators for its satisfaction's dependencies.

The component instantiator, when invoked, will invoke its dependencies' instantiators (if any) and instantiate the object (if necessary). Component instantiation is a two-step process:

1. If the satisfaction to be instantiated is a class (either an implementation or provider class), instantiate the class, using the dependency instantiators to obtain the required components to pass to the class's constructor, injectable setters, and injectable fields.

2. If the satisfaction is a provider (either a provider object or a now-instantiated provider class), invoke the provider's `get()` method.

4.4.8 The Grapht API

Applications embedding Grapht need to be able to do two major things with it:

- Specify policy (bind component types to their respective interfaces).

- Request component instances

```
InjectorBuilder bld = new InjectorBuilder();
bld.bind(I1.class)
   .to(B.class);
bld.within(Left.class, B.class)
   .bind(I2.class)
   .to(C.class);
bld.within(Right.class, B.class)
   .bind(I2.class)
   .to(D.class);

Injector inj = bld.build();
I1 obj = inj.getInstance(I1.class);
assert obj instanceof B;
```

Listing 4.2: Example code to build and use an injector.

Injector is responsible for responding to component requests by resolving their dependencies and instantiating the required components. Injectors are built by an InjectorBuilder, which exposes APIs to bind types to implementations and thereby build up the policy.

Listing 4.2 demonstrates how to construct an injector and configure it with the bindings in fig. 4.6. Bindings are expressed with a so-called 'fluent' API, using chains of method calls to describe a binding in English-like syntax. The final `to` (or `toProvider`) call finishes the binding and adds it to the list of bindings accumulated by the injector builder.

The binding API is built on contexts. By default, bindings go on the root context: they are associated with the context expression '.∗'. The `within` method returns a nested context. Bindings on this context are associated with an expression that matches any context containing the qualified type passed to `within`. That is, for the qualified type (q, τ), it creates a matcher '. ∗ (q, τ).∗'. `within` can be called on its own result, appending additional types to match. The bindings finally added to such a context are applied to any injection context of which the `within` qualifications are a subsequence (each subsequent to `within` appends a

```
bind I1 to B
within (Left, B) {
    bind I2 to C
}
within (Right, B) {
    bind I2 to D
}
```

Listing 4.3: Groovy injector configuration.

another '$(q_i, \tau_i).*$' to the accumulated context expression).

In addition to `within`, the binding API provides `at`, which produces an anchored match by appending the qualified type without a subsequent repeated wildcard. This allows bindings to be restricted so that they only activate for direct dependencies of some type.

LensKit augments the fluent API with a Groovy-based DSL, allowing a more structured configuration style as shown in listing 4.3.

Under the hood, the injector builder is using a binding function builder to produce rule-based binding functions (described in more detail in section 4.4.5), using the resulting binding functions to create a DependencyResolver, and creating a DefaultInjector that uses the configured dependency resolver to resolve components to implementations.

In response to each `getInstance` call, the DefaultInjector asks the dependency resolver to resolve the requested component into a constructor graph. It then re-runs the simplification algorithm, re-using the merge pool from previous instantiations, to continually maintain a single graph of all the components that have been instantiated. This allows instances to be reused between `getInstance` calls as the instance caching policy dictates.

Applications requiring more direct control over Grapht can use the binding function builder and dependency resolver components directly. LensKit does this in order to analyze

and modify constructor graphs before instantiating them.

4.4.9 Providers and Cyclic Dependencies

JSR 330 requires that compliant DI containers support dependencies on providers, not just dependencies on components. That is, if A requires a B, it can express a dependency on a Provider<B> instead. If a provider does not cache instances, injecting it allows components to create multiple new instances of DI-configured dependencies (although we find it to generally be better to use a dedicated factory component for this purposes, to make the intent clearer and to decrease the sensitivity of component behavior to the specific DI configuration). Another major reason for injecting providers is to allow circular dependencies to be instantiated by breaking the cycle of constructor dependencies; the JSR 330 TCK depends on this. In this use case, the DI container supplies a provider as a promise that it will eventually make the object available.

Since Grapht uses providers internally to implement all instantiation, even of classes and pre-instantiated instances, injecting a provider is easy. Supporting cyclic dependencies is somewhat more difficult. When Grapht encounters a dependency on a provider while building the initial solution tree, it doesn't resolve the provider's dependencies immediately. It instead adds the provider to a queue of deferred components to be processed after all non-deferred dependencies have been resolved. Grapht then simplifies the graph of non-deferred components and begins processing the deferred components one by one, simplifying after each, until all dependencies are resolved. Using the simplification phase means that the provider component will depend on the same graph nodes as — and therefore be able to share objects with — other uses of the component, including uses that make the dependency cyclic.

Cyclic dependencies mean that the final object graph will contain cycles that must be

represented in the instantiation plan. Grapht's graph abstraction is restricted to representing rooted DAGs, so the dependency solver maintains a separate list of back edges completing dependency cycles. The instantiator consults the back edge table if it cannot find some required dependency in the graph itself. This keeps Grapht's foundational structures focused on the common case of acyclic dependencies, while allowing for cyclic dependencies if needed. LensKit does not enable Grapht's cyclic dependency support, and can therefore ignore the back edge table (it will always be empty). This has the side effect of preventing LensKit components from depending on providers, but this has not been a problem.

4.5 Grapht in LensKit

Grapht's requirements and design have been driven by the needs of LensKit for configuring and manipulating algorithms. This section describes in more detail how Grapht's features and design enable some of LensKit's sophisticated capabilities.

4.5.1 LensKit's Integration

As mentioned in section 4.4.8, LensKit embeds Grapht by directly using the DependencySolver and working with the resulting constructor graphs. Its LenskitConfiguration object re-exposes the Grapht binding API, using a BindingFunctionBuilder to build up binding functions like Grapht's InjectorBuilder does. It also has a method to configure *root* components; the recommender building process starts at the root components to determine the set of components available, and LensKit does not support the on-the-fly injection supported by Grapht's DefaultInjector. The core LensKit interfaces (section 3.4 are registered as roots by default; if an application wants components other than those transitively available via the dependencies of the selected implementations of the core dependencies to be available in

the final recommender container, it needs to register them as roots.

The LenskitRecommenderEngineBuilder class takes one or more configurations and uses them to build up a recommender engine. It builds a DependencySolver using the binding functions produced by each of the configurations (each configuration contributing 3 binding functions, as described in section 4.4.5), ending the list with the default binding function. It then uses the dependency resolver to compute the constructor graph, which is used by the LenskitRecommender class to instantiate recommender components.

4.5.2 Easy Configuration

Grapht's strong support for defaults, modeled heavily after Guice's capabilities, makes it easy to configure relatively complex networks. We have also wrapped its fluent API in an embedded domain-specific language in Groovy, allowing for very straightforward and readable configuration. A working item-item recommender requires very little explicit configuration:

```
bind ItemScorer to ItemItemScorer
within (UserVectorNormalizer) {
    bind VectorNormalizer to MeanCenteringVectorNormalizer
}
```

4.5.3 Identifying Shareable Components

Computing the object graph as a first-class entity prior to object instantiation allows LensKit to automatically identify objects that can be shared between different uses of the recommender. This is useful in at least two places: identifying common components that can be reused between multiple configurations in an experiment, and pre-building expensive components to be used by multiple threads (or processes) in a running application.

Pre-Building for Web Applications

The core issues that affect the ability for components to be shared across web requests are thread safety and database access. In many common architectures, web applications open database connections or obtain them from a pool on a per-request basis. Any component that needs access to the database therefore needs to be instantiated for each request, so it can access the database connection in use for that request.

Many components, such as the item-item similarity matrix, are both immutable and independent of the database once they have been computed. These components can be instantiated once and shard across requests. Some of these components may require access to the database in at construction time (e.g. to learn a model) but, once constructed, are independent of the database. The goal of LensKit's automated lifecycle separation is to identify and pre-instantiate these components.

The path from LenskitRecommenderEngineBuilder, resolving a LensKit algorithm configuration into an object graph, to the LenskitRecommender that makes its components available is somewhat more complicated than the overview provided in section 4.5.1. In more detail, the LensKit recommender builder does the following:

1. Build a graph from the algorithm configurations.
2. Traverse the graph, looking for shareable components. Each shareable component is instantiated immediately, and its node is replaced as if it were the result of an instance binding.
3. Encapsulate the resulting graph, with shareable components pre-instantiated, in a LenskitRecommenderEngine.

4. LenskitRecommenderEngine allows new LenskitRecommender objects to be created; each such recommender will have a copy of the instantiated graph (sharing the instances of shareable components) and create unique instances of non-shareable components.

In order to support these manipulations, LensKit introduces two annotations to indicate the way components should be handled by the recommender engine builder. One, @Shareable, is applied to components to identify them as candidates for sharing. This annotation marks a component as thread-safe and usable across requests.[8] It must be applied to the implementation, not to the interface; the same interface may well have both shareable and non-shareable implementations.

The second, @Transient, is applied to dependencies of a component to indicate that the dependency is only needed during construction (or, if it is on a provider's dependency, that the provider only needs it to build objects). It promises that, once the constructor or provider has built an object, the object is free of references to the dependency. This allows shareable components to access non-shareable components (such as the data access object) during their construction phases, so long as they do not retain references to them. This is used, for instance, by the item-item model builder on its dependency on the item event DAO: the builder must have access to the data in order to build a similarity matrix, but the matrix will not reference or use the DAO.

In step (2) above, LensKit scans the constructor graph to identify all shareable components that have no non-transient dependencies on non-shareable components. These components are then pre-instantiated and prepared for sharing, and LensKit modifies the

[8] @Shareable is a promise made by the class developer, and they are still responsible to ensure that their class is written to be shareable (among other things, it must be thread-safe). LensKit does not provide any verification that shareable classes actually are shareable. But if the developer has promised that a particular class is shareable, then LensKit will share it unless one of its dependencies precludes sharing.

constructor graph to substitute pre-computed instances constructors in place of the original constructors for the shared components The resulting constructor graph is equivalent to the result of manually pre-instantiating each shared component and using these instances in a configuration. This allows algorithm authors to leverage a lot of framework assistance for deploying their algorithms in realistic environments with just a few Java annotations.

The modified graphs, with the pre-instantiated shared components, can also be serialized to disk and reloaded later. We use this to compute recommender models in a separate process from the web server; once a new model is computed and saved, the web server notices and reloads its version of the model from disk.

The logic that LensKit uses for lifecycle separation could also be adapted to provide robust, automatic support for object scoping in more traditional web applications.

Sharing in Evaluation

The evaluator's use of sharing is somewhat simpler. It processes all the algorithms configured for a evaluation and has Grapht create their component graphs before training or evaluating any of them. It then uses the merge pool to merge all of the graphs, so any component configurations shared by multiple components are represented by common subgraphs.

The evaluator caches shareable objects (identified with the same logic as is used for lifecycle separation) across algorithms. It also establishes dependency relationships between individual evaluation jobs (evaluating a single algorithm on a single data set) so that a single use of a common component is built first, with other uses waiting for it to complete. This is to allow a multithreaded evaluation to start working on other algorithms that do not share common components instead of starting 8 evaluations that will all block on the same model build.

4.5.4 Visualization and Diagnostics

Pre-computing dependency solutions also provides benefits for debugging recommender configurations. First, it ensures that dependency problems fail fast. Since some algorithms have complicated models that, on large data sets, may take a long time to compute, it is useful to fail quickly rather than have dependencies of some component fail only after spending hours computing a model. The multi-stage approach employed by Grapht ensures that all dependency errors are identified as early as possible, as they will result in a failure to build a constructor graph instead of a failure to instantiate needed objects.

It also allows us to provide diagnostic tools such as automatic diagramming of configurations (e.g. fig. 3.3) without incurring the cost of object instantiation. Guice and PicoContainer provide support for inspecting and diagramming configurations, but they both accomplish this by instrumenting the instantiation process. Grapht allows the dependency solution, representing the final object graph, to be computed independently of object instantiation, allowing for cleaner tooling support.

4.5.5 Contextual Policy

The availability of context-sensitive policy in Grapht has influenced the design of LensKit's components and reduced the need for redundant qualifiers. One case where this applies is in the similarity components used by the nearest-neighbor collaborative filters (sections 3.7.4 and 3.7.5). We define specific types for comparing users and items (UserSimilarity and ItemSimilarity, respectively), but many similarity functions, such as cosine similarity, just operate on the vectors. We therefore provide a generic VectorSimilarity class that implements vector similarities without item or user IDs, and provide default implementations of the user- and item-specific similarity functions that delegate to a vector similarity. If both user and item similarities appear in an algorithm configuration, we can use context-sensitive

configuration to select a different vector similarity for each. For example, to use Spearman for comparing users and cosine similarity for items, we can do the following:

```
within (UserSimilarity) {
    bind VectorSimilarity to SpearmanCorrelation
}
within (ItemSimilarity) {
    bind VectorSimilarity to CosineVectorSimilarity
}
```

Without context-sensitive policy, we would need to use a qualifier to distinguish between item and user vector similarities if both are used in the full algorithm. Solely relying on a qualifier, however, would prevent us from configuring *different* item similarities in the same algorithm unless the algorithm components themselves are deeply aware of the composition relationship. With context-sensitive policy, we can just specify enough information to find the location where we want each similarity function. For example, we can configure a hybrid of two differently-configured item-item recommenders:

```
within (Left, ItemScorer) {
    within (ItemSimilarity) {
        bind VectorSimilarity to PearsonCorrelation
    }
}
within (Right, ItemScorer)
    within (ItemSimilarity) {
        bind VectorSimilarity to SpearmanCorrelation
    }
}
```

Context-sensitive policy allows us to achieve these kinds of results and compose individual components into arbitrarily-complex configurations without implementing verbose, error-prone custom wrapper components to expose the particular configuration points

needed as qualifiers. And if we have context-sensitive policy, many qualifiers (such as qualifying the VectorSimilarity dependencies to indicate whether they apply to users or items) become redundant and add no clarity to the design or configuration.

Context-sensitive policy becomes necessary, not just convenient, when configuring hybrid recommenders that reuse the same component implementations in different configurations. Problems of the sort shown in fig. 4.6(a) quickly arise in such algorithms, and context-sensitive policy allows LensKit recommender components to be fully composable into arbitrary graphs with minimum manual configuration.

4.6 The Grapht Model

In this section, we present a formal model of dependency injection that describes a slightly simplified version of Grapht's design and capabilities. The heart of this model is a component request, a request to instantiate the appropriate implementation of some component along with its dependencies. A binding function is used to identify the correct implementation of the request and, recursively, to satisfy its dependencies. All of this happens within a *runtime environment*, defining the universe of discourse in which the injector operates. For a well-formed policy, there will be a unique *constructor graph* produced by using it to satisfy a type request.

This model is useful for defining the specifics of Grapht's algorithms in concise terms, and for reasoning about dependency injection. We use it to show that qualifiers are reducible to contexts and thus strictly less expressive (in section 4.3.3 we demonstrated configurations that cannot be expressed by qualifiers but can with context-sensitive policy).

Our model and its presentation are organized as follows:

1. Define dependency injection component requests, solutions, and policy, with context

and context sensitivity but omitting qualifiers.

2. Show how to solve DI component requests and produce good solution graphs.

3. Extend the definitions with qualifiers and show a reduction from qualified DI to unqualified but context-sensitive DI.

The model and algorithms described in this section are fully independent of any particular language, runtime environment, or object model.

4.6.1 Core Definitions

Dependency injection occurs within the context of a particular *runtime environment*, provided by the language or platform's runtime facilities, libraries in use, and the running application's type definitions.

Definition 1 (Runtime Environment)**.** A runtime environment $\mathfrak{R}$ is a 3-tuple (O, T, C), where

- O is the set of all possible objects.
- T is a non-empty set of types.
- C is a set of constructors. Each constructor $c \in C$ constructs objects of some type τ (denoted $c \rightarrowtail \tau$) and has a set of dependencies expressed as types $\tau' \in T$. The set of dependencies of c is denoted $\mathscr{D}(c)$.

We use set notation to reason about types: if an object $x \in O$ is of type $\tau \in T$, we denote this by $x \in \tau$. Likewise, $\tau \subseteq \tau'$ means that τ is a subtype of τ'.[9]

[9] In a dynamic language, such as JavaScript or Python, T can be considered a singleton set, with qualifiers serving as the sole means of labeling dependencies.

A constructor $c \in C$ for type $\tau \in T$ takes 0 or more input arguments (defined by $\mathscr{D}(c)$) and returns an object of type τ. For the purposes of this model, constructors encapsulate any mechanism for object instantiation; setter and field injection is included in the concept, as are instances and providers (an instance binding results in a nullary constructor that returns the instance, while a provider is a constructor that depends on the provider's dependencies and invokes the provider to create an instance). Some of a constructor's dependencies may be optional; for such a dependency, the injector substitutes a null value if no suitable constructor has been configured. For notational convenience and simplicity, we omit notation for optional vs. mandatory dependencies, but they have negligible impact on the model.

The process of satisfying a component request will result in a *constructor graph*:

Definition 2 (Constructor Graph)**.** A *constructor graph* in a runtime environment $\mathfrak{R}$ is a directed graph G with vertices $\mathrm{V}[G]$ and

- a designated root vertex $\textsc{Root}[G] \in \mathrm{V}[G]$
- a constructor $\mathrm{C}[v] \in C$ associated with each vertex $v \in \mathrm{V}[G]$
- a type $\textsc{Type}[e] \in T$ labeling each edge $e \in \mathrm{E}[G]$

The following properties must also hold:

- All dependencies are satisfied:

$$\forall v \in \mathrm{V}[G].\forall \tau \in \mathscr{D}(\mathrm{C}[v]).\exists v' \in \mathrm{V}[G].\left(\mathrm{C}[v'] \succ \tau \wedge (v \xrightarrow{\tau} v') \in \mathrm{E}[G]\right)$$

- There are no extraneous edges:

$$\forall v \in \mathrm{V}[G].|\{(u, v') \in \mathrm{E}[G] : u = v\}| = |\mathscr{D}(\mathrm{C}[v])|$$

Further, it is useful to define a notion of equality between constructor graphs.

Definition 3 (Constructor Graph Equality)**.** Two constructor (sub-)graphs of G_1 and G_2 rooted at v_1 and v_2 are equal if the following hold:

- $\mathrm{C}[v_1] = \mathrm{C}[v_2]$
- For each $\tau \in \mathscr{D}(\mathrm{C}[v_1])$, let $v_1' \in \mathrm{V}[G_1]$ such that $(v_1 \xrightarrow{\tau} v_1') \in \mathrm{E}[G_1]$ and $v_2' \in \mathrm{V}[G_2]$ such that $(v_2 \xrightarrow{\tau} v_2') \in \mathrm{E}[G_2]$. Then the subgraphs rooted at v_1' and v_2' must also be equal.

G_1 and G_2 may be the same graph, to compare subgraphs with particular roots within the same graph.

In order to support context-sensitive dependency policy, we also define a notion of *context*.

Definition 4 (Context)**.** A context $\chi = \langle c_1, \ldots, c_n \rangle$ is a finite sequence of constructors representing a path from the root of a constructor graph.

The set of all contexts is denoted X; the empty context is $\langle\rangle$. The concatenation of two contexts χ_1 and χ_2 is denoted $\chi_1 + \chi_2$.

Finally, we can define a component request:

Definition 5 (Component Request)**.** A component request $(\tau, \chi) \in T \times X$ is a request for a component of type τ in context χ.

Injection typically begins with a *initial component request* in the empty context: $(\tau_0, \langle\rangle)$.

Component requests are resolved into constructor graphs by means of a binding function:

```
1:  function RESOLVE-DEPS(τ, χ, ℬ)
2:      G ← new empty graph
3:      c ← ℬ(τ, χ)
4:      if c is undefined then
5:          fail there is no binding for τ in χ
6:      v ← new vertex in G
7:      C[v] ← c
8:      for τ′ ∈ 𝒟(c) do
9:          G′ ← RESOLVE-DEPS(τ′, χ ⤛ ⟨c⟩, ℬ)
10:         V[G] ← V[G] ∪ V[G′]
11:         E[G] ← E[G] ∪ E[G′] ∪ {v —τ′→ ROOT[G′]}
12:     return G
```

Listing 4.4: Resolving component request dependencies.

Definition 6 (Binding Function)**.** A binding function $\mathscr{B} : T \times X \to C$ is a partial function such that for each (τ, χ) where $\mathscr{B}(\tau, \chi)$ is defined, $\mathscr{B}(\tau, \chi) \succ \tau$.

To implement the dependency resolution task of a dependency injector, therefore, we can use a binding function $\mathscr{B}$ to resolve the (recursive) dependencies of an initial type request $(\tau, \langle\rangle)$. This process yields a constructor graph G that can be used to directly instantiate the required components. The details of how this to compute a constructor graph from a binding and a type request is the subject of the next section.

4.6.2 Resolving Component Requests

In order to satisfy a component request, we must use the binding function (policy) to compute a constructor graph that, when instantiated, will produce an instance of the desired component with all of its dependencies. We want implement the transformation of fig. 4.2(a) into one of the object diagrams in fig. 4.2(b).

The RESOLVE-DEPS function in listing 4.4 naïvely produces a constructor graph to satisfy

a component request (τ, χ) with a policy $\mathscr{B}$. This algorithm works — and is the core of Grapht's dependency resolution in practice — but has two key deficiencies as written. First, it loops endlessly if there cyclic dependencies.

Second, it produces a fresh vertex for every dependency relation encountered, rather than reusing vertices representing the same components. The constructor graph it produces, while meeting the requirements for an instantiable constructor graph, is a tree. Many applications, including LensKit, want to reuse components if they are required by multiple other components rather than creating a new instance of a component each time it is required. This can be achieved in two ways: either the instantiator can memoize its instantiation process, so that applying the same constructor to the same dependencies reuses the existing instance, or the graph can be simplified by collapsing subgraphs representing the same component configuration into a single subgraph that has multiple incoming edges on its root node. The latter method has the advantage of encoding possible component reuse in the graph itself, making it available to static analysis operating on the graph, while not precluding the instantiator from creating multiple instances if policy so dictates. Vertices can even carry additional attributes specifying such policy decisions.

Therefore, we want to produce constructor graphs with *maximal reuse*: each constructor appears on exactly one vertex for each unique transitive dependency configuration to which it applies. We also want to detect and fail in the face of cyclic dependencies.

If the binding function is context-free ($\forall \tau, \chi . \mathscr{B}(\tau, \chi) = \mathscr{B}(\tau, \langle\rangle)$), then it is easy solve both of these problems: Resolve-Deps can be adapted into a depth-first graph traversal that seen set to detect cycles and a memoization table to reuse constructor graphs. Resolve-Deps-XFree (listing 4.5) shows such an algorithm.

If $\mathscr{B}$ is not context-free, the situation is more complicated. The arguments passed to a constructor may vary based on the context in which that constructor is used. We say that

```
 1: function Resolve-Deps-XFree(τ, χ, ℬ)
 2:     G ← new graph
 3:     XFree-Dep-Traverse(τ, χ, ℬ, G, {}, {})
 4:     return G
 5: function XFree-Dep-Traverse(τ, χ, ℬ, G, M, S)
 6:                                          ▷ M : C → V[G] is constructor map
 7:                                          ▷ S ⊆ C is set of seen constructors
 8:     c ← ℬ(τ, χ)
 9:     if c is undefined then
10:         fail there is no binding for τ in χ
11:     else if M[c] is defined then
12:         return v
13:     else if c ∈ S then                              ▷ seen but not finished
14:         fail cyclic dependency of constructor c
15:     else
16:         v ← new vertex in G
17:         C[v] ← c
18:         add c to S
19:         for τ′ ∈ 𝒟(c) do
20:             v′ ← XFree-Dep-Traverse(τ′, χ ⧺ ⟨c⟩, ℬ, G, M, F)
21:             add v --τ′--> v′ to E[G]
22:         M[c] ← v
23:     return G
```

Listing 4.5: Context-free resolution.

such a constructor's dependencies are *divergent*. A constructor may have divergent dependencies even if it has the same bindings for all its direct dependencies due to divergence in its transitive dependencies, as in fig. 4.6. This means that it is difficult, if not impossible, to determine whether a constructor's dependencies will be divergent. Further, it is technically possible to have several repetitions of a constructor before breaking a cycle. The following configuration is an example of such a situation:

$$c \succ \tau \tag{4.1}$$

$$\mathscr{D}(c) = \{\tau\} \tag{4.2}$$

$$c' \succ \tau \tag{4.3}$$

$$\mathscr{D}(c') = \{\} \tag{4.4}$$

$$\mathscr{B}(\tau, \chi) = \begin{cases} c' & \text{if } \chi = \langle c, c, c \rangle \\ c & \text{otherwise} \end{cases} \tag{4.5}$$

This configuration resolves to the constructor graph $c \xrightarrow{\tau} c \xrightarrow{\tau} c \xrightarrow{\tau} c'$. However, when resolving τ to c the second and third time, the resolution algorithm has no way of knowing whether the cycle will terminate or not. While this particular example is degenerate, if c has additional dependencies, such stacked configurations are not difficult to envision.

Grapht uses an efficient two-phase approach for producing constructor graphs with maximal reuse while using context-sensitive policy. It first applies $\mathscr{B}$ and resolves dependencies to produce a constructor tree using Resolve-Deps, augmented with a depth limit on the dependency tree to catch cyclic dependencies and ensure termination. It is possible for the policy to produce acyclic but very deep graphs, but in practice it is unusual to have extremely deep object graphs. This approach results in an algorithm that has an arbitrary but tunable limit instead of a structural limitation such as prohibiting a constructor from being used to satisfy one of its own dependencies.

After producing a constructor tree, we simplify the tree by detecting and merging all identical subgraphs. Listing 4.6 shows an efficient dynamic programming algorithm to perform this simplification; this is the algorithm used by MergePool (section 4.4.3). The resulting graph has a single vertex for each combination of a constructor and its transitive

```
1:  function SIMPLIFY-GRAPH(G)
2:      ⟨v_1, ..., v_n⟩ ← TOPO-SORT(G)[a]
3:      G′ ← new empty graph
4:      ⟨v′_1, ..., v′_n⟩ ← new list
5:      m ← new map C × 𝒫(V[G′]) → [1, n]
6:                              ▷ map constructors and dependency vertices to list positions
7:      for i ← 1 to n do
8:          D_i ← {v′_j : (v_i, v_j) ∈ E[G]}        ▷ ∀v′_j ∈ D_i, j < i and thus already merged
9:          c_i ← C[v_i]
10:         if m(c_i, D_i) is defined then                          ▷ reuse previous vertex
11:             j ← m(c_i, D_i)
12:             v′_i ← v′_j
13:         else                                ▷ constructor × deps unseen, add new vertex
14:             v′_i ← v_i
15:             add v′_i to V[G′]
16:             for v′_j ∈ D_i do
17:                 add (v′_i, v′_j) to E[G′]
18:             m(c_i, D_i) ← i
19:     ROOT[G′] ← v′_n
20:     return G′
```

[a] TOPO-SORT topologically sorts a DAG so that edges go from right to left ($(v_i, v_j) \in \mathrm{E}[G] \implies j < i$) and $v_n = \mathrm{ROOT}[G]$.

Listing 4.6: Simplify Constructor Graph

dependencies in the solution, but may use the same constructor for multiple vertices, thus achieving maximal reuse without sacrificing any generality. SIMPLIFY-GRAPH starts at the leaves of the tree and merging all possible vertices prior to merging the constructors that may depend on them.

That SIMPLIFY-GRAPH produces graphs with maximal reuse can be shown with strong induction:

Base case (v_1)

$\mathrm{C}[v_1]$ will have no dependencies; otherwise, v_1 would have outgoing edges and would not be the first node in the topological sort. Also, the subgraph rooted at v_1 has

maximal reuse among the vertices seen so far: the algorithm has not encountered any other vertices, so there are no other vertices with which it might be redundant. Therefore, adding it to the currently-empty G' means G' has maximal reuse.

Inductive step (v_i, $i > 1$)

G' has maximal reuse. Each vertex v_j for $j < i$ has a corresponding vertex $v'_j \in \mathrm{V}[G']$ that is the root of a non-redundant subgraph of G'. The algorithm takes maps the dependencies of v_i in G to their corresponding vertices in G' (line 8), and looks up the constructor and this resolved dependency set in the table of nodes seen so far. If the constructor has already been applied to equivalent dependencies, then it will appear in the lookup table, and its resulting vertex (and subgraph) will be used, maintaining maximal reuse. If (c_i, D_i) does not appear in m, then either c_i has never been seen, or at least one vertex has a different configuration (otherwise, it would be in m). A new vertex is generated for this unique configuration and maximal reuse is preserved.

Since the singleton graph trivially has maximal reuse, and no subsequent iteration breaks the maximal reuse property, the final graph G' will have maximal reuse.

4.6.3 Bindings

We have so far ignored how the binding function $\mathscr{B}$ is represented. The high-level model we employ is independent of any particular binding function representation, so an alternative mechanism could be substituted here while retaining the definitions and algorithms of sections 4.6.1 and 4.6.2 This section describes the policy representation used by Grapht.

Grapht's binding function is built a set of individual *bindings* specified by the application's configuration, each of which binds a contextually-qualified type to either a constructor or another type.

Definition 7 (Binding)**.** A *binding* $b = (\bar{\tau}, \tilde{\chi}) \mapsto t$ specifies the implementation to use, where

- $\bar{\tau} \in T$ is a type to match.
- $\tilde{\chi}$ is a *context expression*, a regular expression over contexts.
- t is a *binding target*, either a type or a constructor. The type resulting from t is denoted $\tau(t)$; $\tau(t) = t$ if t is a type, and $\tau(t) = \tau'$ if t is a constructor $c \succ \tau'$.

Evaluating $\mathscr{B}(\tau, \chi)$ is a matter of finding the matching binding $(\bar{\tau}, \tilde{\chi}) \mapsto t$ where $\bar{\tau} = \tau$ and $\tilde{\chi}$ matches χ. If t is a constructor, then $\mathscr{B}(\tau, \chi)$; if t is a type, then the bindings are evaluated recursively as $\mathscr{B}(t, \chi)$.

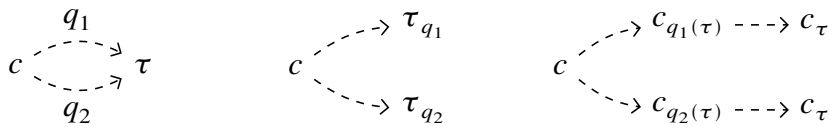
Any reasonable predicate over constructors can serve as the basis for atoms (defining the set of constructors that will be matched at a particular position) in context expressions. In runtime environments with subtyping, types are a reasonable choice of atom, with a type τ matching any constructor $c \succ \tau'$ where $\tau' \subseteq \tau$; a root type can serve as the wildcard.

If there are multiple bindings that match (τ, χ), then the *most specific* is selected. The exact definition of specificity is dependent on the details of the runtime environment and type system; section 4.4.5 describes the specificity notion used in Grapht.

4.6.4 Qualifiers

So far, our model has ignored qualifiers. In this section, we extend it with qualifiers and show how to eliminate qualifiers to reducing to context-sensitive policy, outlining a proof that context-sensitive policy is strictly more expressive than qualifiers.

As discussed in section 4.3.2, qualifiers are a convenient means of distinguishing between dependencies of the same type, particularly when a single component has multiple



(a) Original (b) Transformed dependencies (c) Solution (unsimplified)

Figure 4.9: Reduction of qualified graph.

such dependencies They are also useful for categorizing dependencies in dynamically-typed environments such as Python.

To add qualifiers to our model, we make the following amendments:

- Augment the runtime environment with a set of qualifiers Q. Qualifiers can be considered to be opaque labels. We designate a particular qualifier $q_\perp$ to represent the lack of a qualifier.
- Constructor dependencies now have qualifiers: $\mathscr{D} : C \to Q \times T$
- Component requests now have qualifiers and are expressed as triples (q, τ, χ).
- Binding functions now take qualifiers: $\mathscr{B}(q, \tau, \chi)$.
- Constructor graph edges are labeled with qualified types ($\mathrm{E}[G] \subseteq Q \times T$).
- Contexts are now sequences of qualified constructors $\chi = \langle (q_1, c_1), \ldots, (q_n, c_n) \rangle$, where q_i is the qualifier on the edge leading to the vertex labeled with c_i.

The resolution algorithms simply need to pass the qualifier associated with each dependency to the binding function and associate the qualifiers with the correct edge labels.

Figure 4.9 is a graphical depiction of the reduction of a constructor c with two qualified dependencies on τ.

There are three steps to reducing qualifiers to context matching:

1. Replace each qualified dependency (q, τ) with a dependency on a synthesized type $\tau_q \subseteq \tau$. These synthetic types may be realized as actual types in the runtime environment, or they may exist only as bookkeeping entities in the DI container.

2. Modify the initial component request (q, τ, χ) to be $(q_\perp, \tau_q, \chi)$ if $q \neq q_\perp$.

3. Modify the bindings as follows:

   - Bind each synthetic qualifier type τ_q to a constructor $c_{q(\tau)} \rightarrowtail \tau_q$ with $\mathcal{D}(c_{q(\tau)}) = \langle \tau \rangle$.
   - For each binding $b = (\bar{q}, \bar{\tau}, \tilde{\chi}) \rightarrow t$, substitute the binding $(\bar{\tau}, \tilde{\chi} \nplus \langle \tilde{q} \rangle) \rightarrow t$, where $\tilde{q}$ matches any synthetic constructor $c_{q(t)}$ whose corresponding qualifier is matched by $\bar{q}$.

We show how to modify bindings; any computable binding function should be able to be similarly modified to look for contexts terminating in t_q.

This reduction works by replacing each qualified dependency (q, τ) resulting in a constructor c with a constructor chain $c_{q(t)} \xrightarrow{\tau} c$. Any policy that examines the qualifier attached with a dependency type can instead look at the context to see if the type is being configured to satisfy the dependency of a synthetic constructor.

After this reduction, the only qualifier in use is $q_\perp$, and the only qualifier matcher is $\top$, so qualifiers can be removed entirely. Not only are qualifiers unneeded, but they do not add any expressive power over context-sensitive policy.

4.7 Conclusion

This chapter has described an approach to dependency injection based on a mathematical model of dependencies and their solutions and a Java implementation using this framework.

Grapht provides static analysis capabilities, context-sensitive policy, and extensive defaulting capabilities, allowing it to better meet the needs of LensKit than the existing solutions.

Our approach to context-sensitive policy allows expressive matching on deep context with easy configuration. For configuring recommender applications, this allows LensKit's individual components to be reconfigured into arbitrarily complex hybrid configurations, allowing extensive code reuse. One of LensKit's design goals is to provide an extensive collection of building blocks that can be combined into sophisticated algorithms, and the ability to configure them without requiring extensive and verbose object instantiation code is crucial to that aim. We have also shown that context-sensitive policy is strictly more powerful than the dependency qualifiers provided by many current dependency injection frameworks; while we expect that qualifiers will live on due to their convenience, they can be viewed as a syntax sugar on top of a more expressive paradigm.

There are a variety of extensions to dependency injection that may be worth considering in the future. One is *weighted dependency injection*: under this scheme, constructors or bindings have associated weights expressing the cost of using them, and the injector tries to find the lowest-cost solution to the component request. This problem is likely NP-hard.

Opportunistic dependency injection is a simplified extension that is likely more practical. In opportunistic DI, some optional dependencies are marked as "opportunistic", meaning that they will only be instantiated and used if required by some other component as a non-opportunistic dependency. They differ from normal optional dependencies in that an optional dependency will be supplied if it is possible to satisfy the dependency given the binding function, while an opportunistic dependency is only supplied if the configured constructor is invoked to satisfy some other dependency in the DIP. The key use case for this extension is when a component A can operate more efficiently if an expensive component B is available, but the efficiency gain alone is not sufficient to warrant the cost of instantiating

B. If some other component requires B, however, then A can take advantage of it under opportunistic DI. In LensKit this comes up with some of the data structures used for iterative training of models such as the FunkSVD model. The structures used to make the FunkSVD model training process efficient can be used by many other components to decrease time and memory requirements, but it is not worth the cost of computing them just to compute the mean of the ratings in the system.

Grapht has proven to be a valuable tool in making LensKit flexible and easy to use. We hope that its well-defined model and straightforward implementation will make it a useful platform for future developments in dependency injection.

Chapter 5

Configuring and Tuning Recommender Algorithms

This chapter describes several offline experiments we have run using LensKit. These experiments serve two primary purposes: to improve our understanding of the behavior of different algorithms and algorithm configurations, and to validate LensKit through reproducing and extending previous results. The diversity of experiments we present here, and their accompanying source code, also demonstrate the flexibility and usefulness of LensKit for a variety of recommender research tasks, as well as being independent research contributions in their own right.[1]

We first present comparative evaluation of several design decisions for collaborative filtering algorithms in the spirit of previous comparisons within a single algorithm [HKR02; Sar+01]. We examine LensKit's user-user, item-item, regularized gradient descent SVD algorithms. These experiments extend previous comparative evaluations to larger data sets and multiple algorithm families and serves to demonstrate the versatility of LensKit and its capability of expressing a breadth of algorithms and configurations. In considering some configurations omitted in prior work we have also found new best-performers for algorithmic choices, particularly for the user-user similarity function and the normalization for cosine similarity in item-item CF. This set of experiments serves to show LensKit's versatility in recommender experimentation, and fill in gaps in our current understanding of how to

[1] This work was done in collaboration with Michael Ludwig, Jack Kolb, Lingfei He, John T. Riedl, and Joseph A. Konstan. Portions have been published in [Eks+11]; other portions are currently in preparation. Jack Kolb and Lingfei He were particularly involved in the work on tuning baselines and item-item CF.

tune and configure commonly-used collaborative filtering algorithms. These experiments also provide insight into possible strategies for systematically tuning recommender system parameters (or the difficulties of doing so, in the case of FunkSVD.

We conclude this chapter with some results on the impact of rank-based evaluation on recommender configuration and design.

5.1 Data and Experimental Setup

These experiments use several common data sets:

ML-100K The MovieLens 100K data set, consisting of 100K user ratings of movies from the MovieLens movie recommendation service.

ML-1M The MovieLens 1M data set.

ML-10M The MovieLens 10M data set. This data set also has 100K 'tag applications', events where users apply a tag to a movie.

Y!M The Yahoo! Music data set, containing user ratings of songs on the Yahoo! Music service and made available through the Yahoo! WebScope program. Unlike the MovieLens data sets, which have a single file of rating data, this data set is pre-split into 9 train/test segments. We do not re-combine the data, but use each train-test split as-is from Yahoo!.

Y!M Subset A subset of one of the training sets in the Yahoo! Music data set. The subset was produced by sampling 10% of the items and retaining all their ratings. We use a subset so that we can experiment with the sparser domain while maintaining reasonable experimental throughput. LensKit is capable of running on the full data set,

but it takes substantial time to build and evaluate such models, making it difficult to conduct extensive experiments.

Table 5.1 summarizes the size and sparsity of these data sets.

Data Set	Range	Ratings	Users	Items	$\|R\|/\|U\|$	$\|R\|/\|I\|$	Density
ML-100K	$[1,5]/1$	100,000	943	1682	106.04	59.45	6.305%
ML-1M	$[1,5]/1$	1,000,209	6040	3706	165.60	269.89	4.468%
ML-10M	$[0.5,5]/0.5$	10,000,054	69,878	10,677	143.11	936.60	1.340%
Y!M	$[1,5]/1$	717,872,016	1,823,179	136,736	393.75	5250.06	0.288%
Y!Music	$[1,5]/1$	7,713,682	197,930	13,673	38.97	564.15	0.285%

Table 5.1: Rating data sets

Most of our results are using the ML-100K and ML-1M data sets. ML-100K allows us to directly replicate and compare with prior work, while ML-1M provides significantly more data while being small enough for good experimental throughput. We also ran some configurations on ML-10M and Yahoo! Music. Unless otherwise specified, all charts are over the ML-1M data set.

For each data set, we performed 5-fold cross-validation with LensKit's default method described in section 3.8.2. 10 randomly-selected ratings were withheld from each user's profile for the test set, and the data sets only contain users who have rated at least 20 items.

The Y!M data set is distributed by Yahoo! in 10 train-test sets, with each test set containing 10 ratings from each test user; we used the provided train/test splits and do not re-crossfold for this experiment.

For each train-test set, we built a recommender algorithm and evaluated its predict performance using MAE, RMSE, and nDCG.

5.2 Baseline Scorers

The baseline scorers described in section 3.7.3 are a critical part of many LensKit recommender configurations. The provide fallback scores when a collaborative filter does not have sufficient data to score an item, and are the basis for many standard normalization techniques. They can also be surprisingly effective rating predictors in their own right. This section documents the relative performance of LensKit's different standard baseline scorers, and the behavior of the Bayesian damping term that they support.

We consider four baseline predictors:

- Global mean rating (μ)
- User mean rating ($\mu + \hat{\mu}_u$, where $\hat{\mu}_u$ is the user's mean offset from the global rating; $\hat{\mu}_u = 0$ for users with no ratings)
- Item mean rating ($\mu + \hat{\mu}_i$, where $\hat{\mu}_i$ is computed for items as $\hat{\mu}_u$ is for users)
- Item-user personalized mean ($\mu + \hat{\mu}_i + \tilde{\mu}_u$, where $\tilde{\mu}_u$ is the user's mean offset from item mean for each of their ratings)

Figure 5.1 shows the RMSE of each of these algorithms on several of our data sets. Since the user mean rating does not rank items (all items will have the same rank for a given user), nDCG is not interesting for this comparison. The relative performance of item and user mean is inverted between the ML-1M and Y!Music data sets; this may be a function of the differing sparsities in the user and item dimensions.

LensKit's baseline scorers also support a mean damping term to bias means towards a neutral value until there are sufficient ratings to have a good sample of the user or item's bias. A damping term of N is equivalent to assuming that the user or item has N ratings at

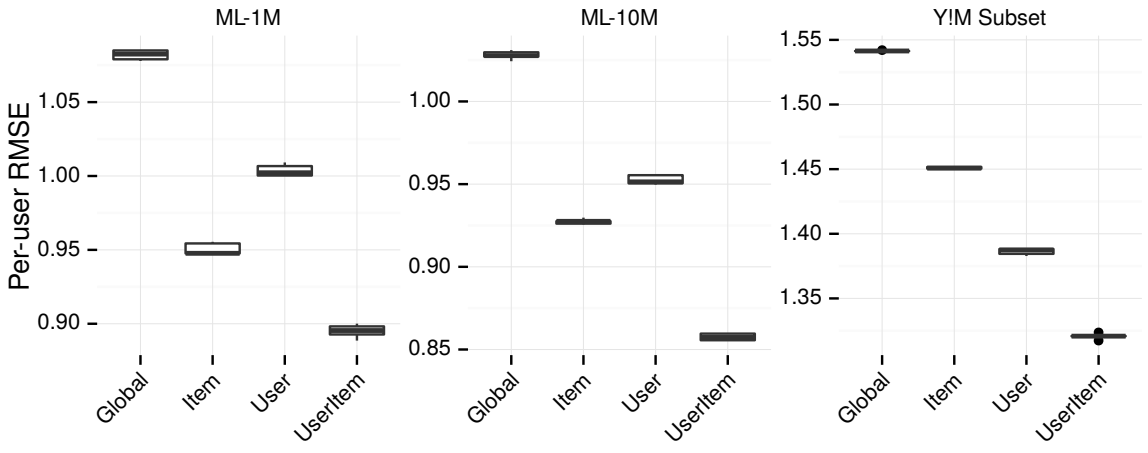


Figure 5.1: Baseline scorer accuracy

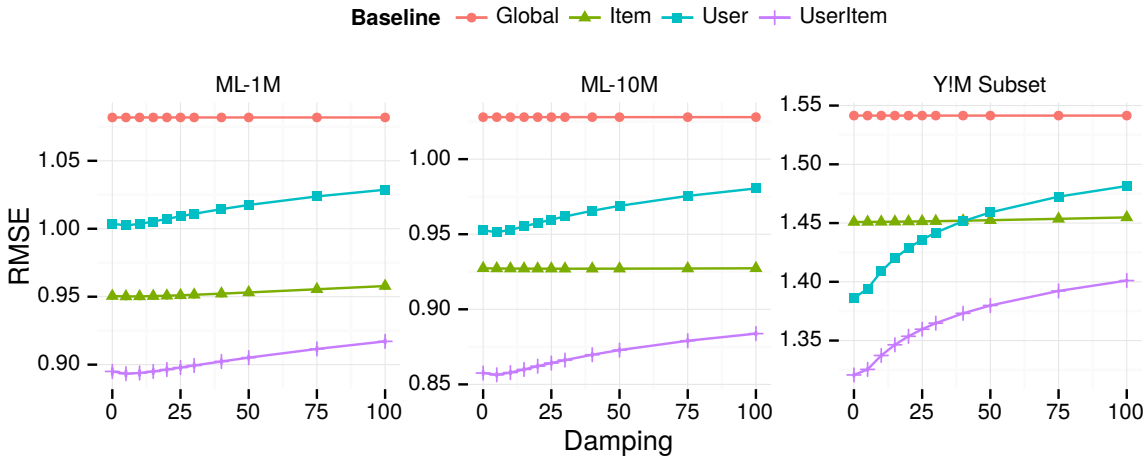


Figure 5.2: Impact of baseline damping

the global average value (for the user-item mean, the damping on a user assumes N ratings at the item average, not global average). Figure 5.2 shows the effect of damping on the baseline predictor's accuracy. A small amount of damping provides negligible benefit in predicting for MovieLens with user means and no benefit on Y!Music. When using a rank accuracy metric (not shown), damping provides no benefit; this is likely due to its lack of impact on item means, since adjusting the user mean does not affect the ranking of items for a user. It does not seem worth the cost to tune damping; either omitting mean damping or setting it to a small value (e.g. 5) and leaving it appear to be reasonable and defensible decisions.

5.3 User-User CF

Herlocker, Konstan, and Riedl [HKR02] tested a variety of configuration choices for user-user collaborative filtering to assess their relative performance. LensKit's flexibility allows us to revisit this work and extend it with additional configurations not considered as well as new metrics (the original work considered only MAE).

Two of the key configuration decisions in user-user CF are the choice of similarity function for comparing users and the number of neighbors to use in each prediction. Herlocker, Konstan, and Riedl [HKR02] tested Pearson and Spearman correlations with and without significance weighting (a damping term to reduce the similarity of users with few rated items in common). We consider both of these configurations, as well as cosine vector similarity over both the raw ratings and the mean-centered ratings.

Figures 5.3 and 5.4 show the performance of user-user CF on the ML-100K and ML-1M data set for these similarity functions over several neighborhood sizes with different choices of normalization strategy prior to averaging for the final prediction. *User* scores items with

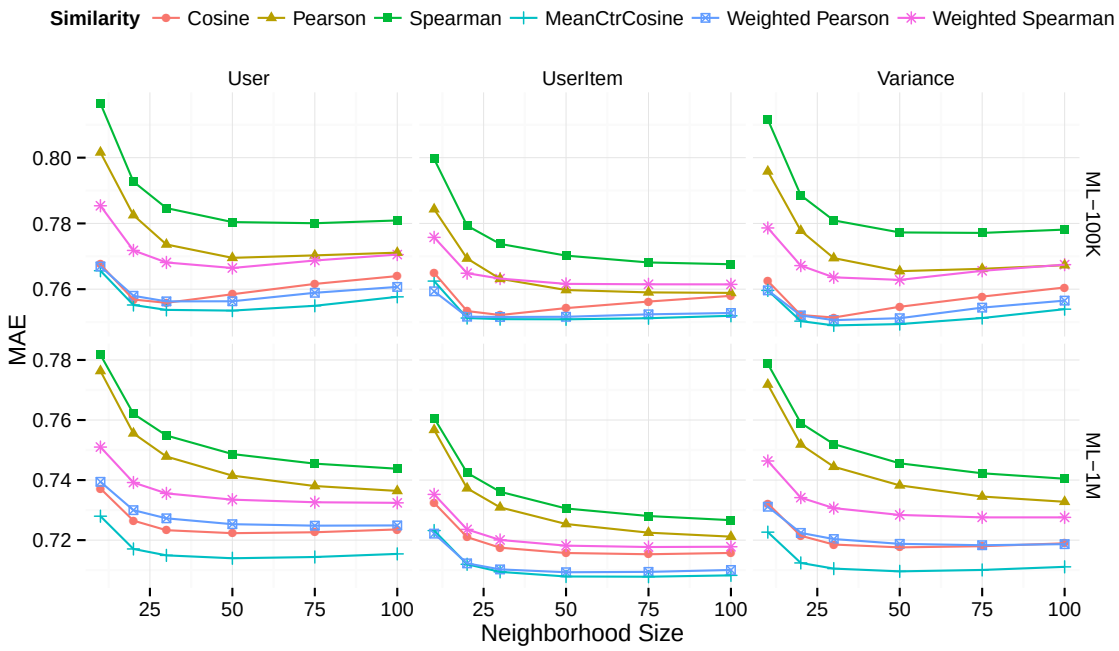


Figure 5.3: Prediction accuracy (MAE) for user-user CF, across two data sets with different choices of similarity function and score normalization.

a weighted average over deviations from each user's mean rating, *UserItem* averages over deviations from the user-item mean, and *Variance* z-normalizes user ratings prior to scoring. This is integrated into the scoring function as follows:

$$p_{ui} = f^{-1}\left(\frac{\sum_{v \in \mathcal{N}} \text{sim}(u,v) f(r_{vi})}{\sum_{v \in \mathcal{N}} |\text{sim}(u,v)|}\right)$$

$$f_{\text{User}}(r_{ui}) = r_{ui} - \mu_u$$

$$f_{\text{UserItem}}(r_{ui}) = r_{ui} - \mu - b_i - b_u$$

$$f_{\text{Variance}}(r_{ui}) = \frac{r_{ui} - \mu_u}{\sigma_u}$$

For items for which user-user could not generate recommendations, we used the user-item mean; all nonpositive similarities were excluded. The Weighted versions of the Pearson

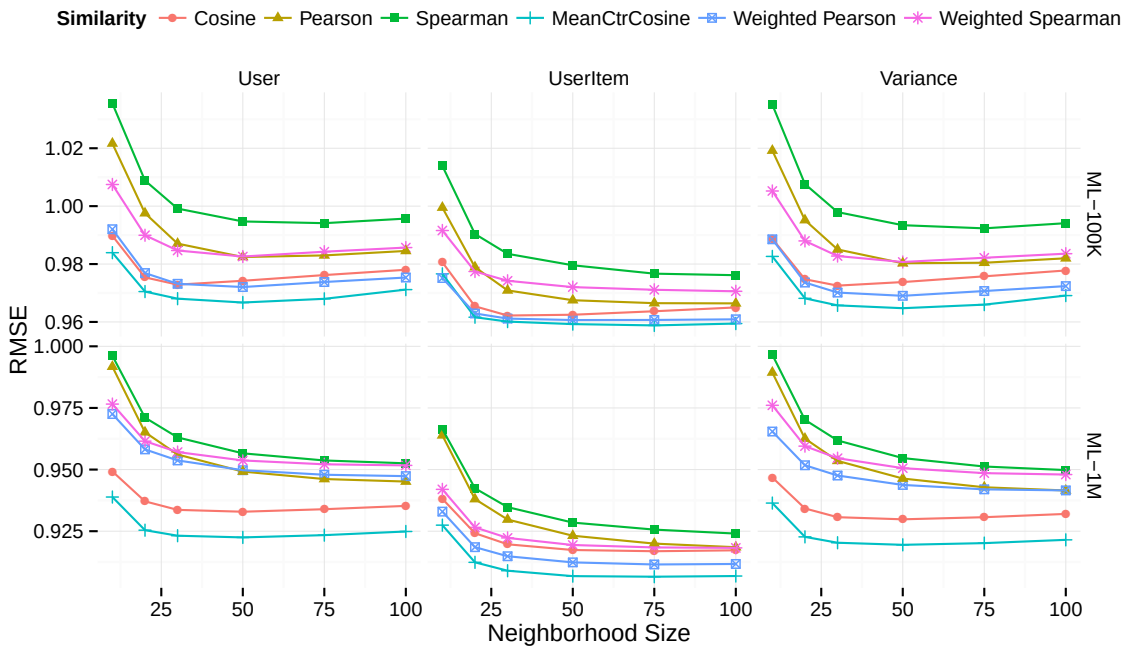


Figure 5.4: Prediction accuracy (RMSE) for user-user CF, across two data sets with different choices of similarity function and score normalization.

and Spearman correlations use significance weighting with a threshold of 50 to reduce the similarity of users with few rated items in common. Significance weighting multiples the similarity by $\min(|I_u \cap I_v|, 50)/50$, decreasing the influence of neighbors until they have 50 items in common.

Our ML-100K results differ slightly from those reported by Herlocker *et al.* — they found weighted Spearman and Pearson to perform equivalently, while we Pearson to consistently outperform Spearman — but our results are consistent with their recommendation to use Pearson instead of Spearman, and are otherwise similar. We have tried several different configurations to attempt to recreate the exact results; our inability to do so demonstrates the need for an improved culture of reproducibility in recommender systems research.

Of particular note is the performance of cosine similarity, which they did not consider.

MeanCtrCosine first normalizes each user's ratings by subtracting their mean rating, then computes the cosine vector similarity between them. Early work that found cosine to perform poorly for user-user CF [BHK98] did not explicitly consider mean-centering data prior to computing the similarity.

There is also another way of looking at this relationship. If the two users have rated the same set of movies, cosine similarity is mathematically equivalent to the Pearson correlation:

$$\begin{aligned}
\cos(\hat{\mathbf{u}}, \hat{\mathbf{v}}) &= \frac{\hat{\mathbf{u}} \cdot \hat{\mathbf{v}}}{\|\hat{\mathbf{u}}\|\|\hat{\mathbf{v}}\|} \\
&= \frac{\sum_i \hat{u}_i \hat{v}_i}{\sqrt{\sum_i \hat{u}_i^2}\sqrt{\sum_i \hat{v}_i^2}} \\
&= \frac{\sum_i (u_i - \mu_u)(v_i - \mu_v)}{\sqrt{\sum_i (u_i - \mu_u)^2}\sqrt{\sum_i (v_i - \mu_v)^2}} \\
&= \operatorname{cor}(u, v)
\end{aligned}$$

However, the two users will not have rated the same movies (if they did, then the neighbor would not be useful, because it cannot contribute information about any new items). Historically, the Pearson correlation has been computed by restricting all sums to be over the items the users have both rated ($I_u \cap I_v$). This is consistent with the general statistical application of correlation. Cosine similarity has sometimes been implemented this way, but in LensKit and historical research [BHK98] the sum is effectively over the union of both item sets ($I_u \cup I_v$), treating missing values in the normalized vectors as 0. This means that, for users who have not rated many of the same items, the numerator decreases (since there are fewer nonzero rating products to sum) but the denominator may still be large (since it considers all items rated by each user). As a result, the similarity between users who have each rated many items but not many in common is naturally discounted, roughly proportional to

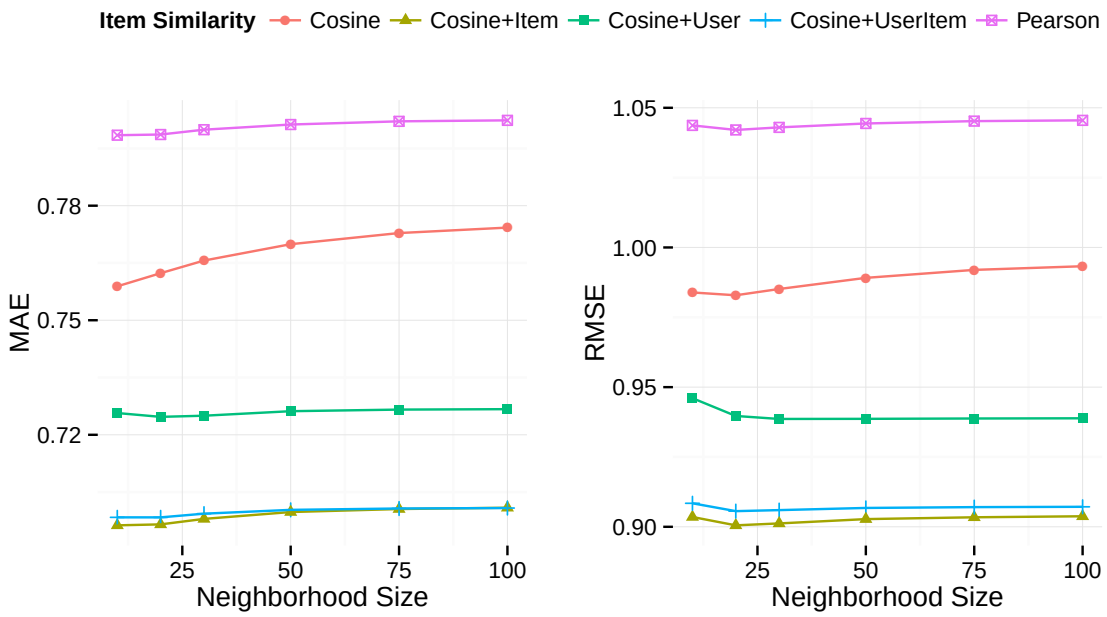


Figure 5.5: Prediction accuracy for item-item CF on ML-1M.

$|I_u \cap I_v| / (\sqrt{|I_u|}\sqrt{|I_v|})$ (the relationship is not linear due to its dependence on the actual values of the ratings, not just their presence). We can view this natural discounting as a parameter-free version of significance weighting, and our results here show that it seems to be just as effective, if not superior.

Another configuration not considered in previous published user-user literature is averaging over deviations from the full user-item personalized mean. Most work has focused on mean-centering or z-normalizing user vectors prior to computing predictions. We observe here that subtracting the user-item mean — so the user-user collaborative filter is only attempting to model the deviation of each rating from the user and item biases — outperforms both approaches that only consider the user's ratings.

5.4 Item-Item CF

Figure 5.5 summarizes the performance we achieved with item-item CF (section 3.7.4), revisiting two of the configuration dimensions explored by Sarwar et al. [Sar+01]. The neighborhood size is the number of neighbors actually considered for each prediction; in all cases, the computed similarity matrix was truncated to 250 neighbors per item. No significance weighting or damping was applied to the similarity functions. Each of the different cosine variants reflects a different mean-subtracting normalization applied prior to building the similarity matrix; user-mean cosine corresponds to the adjusted cosine used by Sarwar et al. Consistent with that work, normalized cosine performs the best, and this result still holds on the larger data set. We also find that normalizing by item mean performs better than user mean; this suggests that measuring similarity by users whose opinion of an item is above or below average provides more value than measuring it by whether the prefer the item more or less than the average item they have rated. This result is quite surprising, which is quite possibly why it has not been tried in prior work. However, with LensKit's flexibility, we decided to try all the baseline normalizers, and found it.

Similarity function and neighborhood size are just a few of the configuration points LensKit's item-item implementation exposes, however. Other parameters that affect item-item's performance and behavior include:

- The number of similar items to retain in the model
- Item similarity damping
- Normalization strategy

The similarity damping term β is a parameter to bias the similarity of items with few users in common towards 0, reflecting the lack of information about their true similarity. For

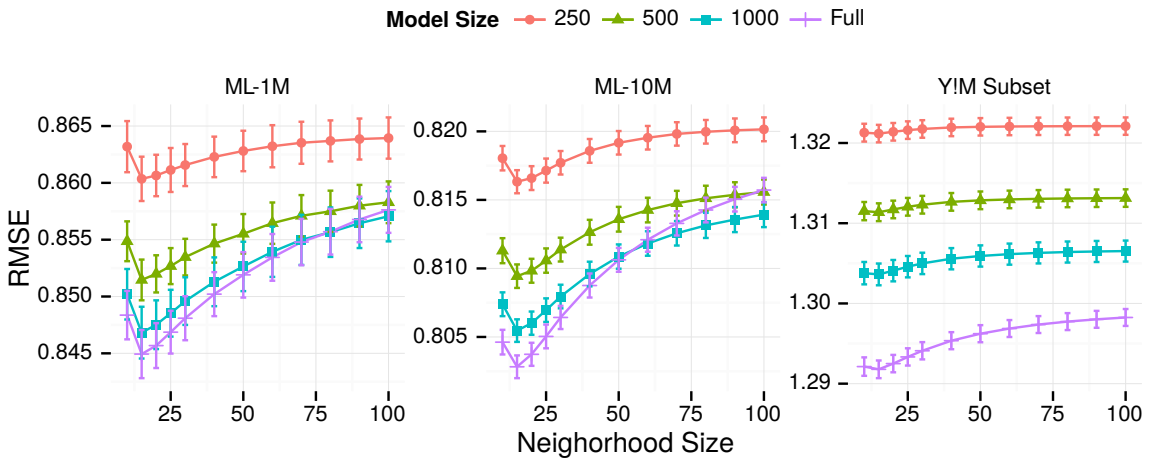


Figure 5.6: Item-item accuracy by neighborhood size.

cosine similarity, it is incorporated in the denominator ($sim(i,j) = \frac{\vec{r}_i \cdot \vec{r}_j}{\|\vec{r}_1\|_2 \|\vec{r}_2\|_2 + \beta}$). It achieves the same goal as significance weighting [HKR02] in an arguably more elegant manner.

Parameter Relationships

To develop a systematic method of efficiently tuning item-item CF, we want to identify the relationships between parameters. In particular, we want to identify interaction effects between parameters with respect to to error metrics in order to see whether some parameters can be trained independently. If the optimal choice for one parameter does not affect the optimal choice for another, then those parameters can be disentangled and trained independently instead of relying on grid search. This decreases the parameter search space for those parameters from $O(mn)$ to $O(m+n)$.

Figure 5.6 shows the accuracy of item-item CF for different model sizes as the neighborhood size is varied. For this evaluation, we normalized ratings by subtracting the item mean, used no similarity or baseline damping, and used item mean for fallback predictions. We observe two key things from this chart:

- Relatively few neighbors are needed (10–20 is a reasonable value across the board).
- There is no significant interaction between model size the optimal value of the neighborhood size. The curve adjusts slightly for different model sizes, but does not affect the optimal neighborhood size.

Since they do not interact within a reasonable range of neighborhood sizes, neighborhood size and model size can be picked independently to achieve an optimal combination. This is expected theoretically: since models and neighborhoods are chosen by the same criterion (similarity), the only difference that the model size makes is restricting the available neighbors. For any prediction where there are enough neighbors in the model for a full neighborhood, having additional neighbors in the model provides no additional benefit.

Figure 5.7 shows accuracy as the similarity damping value is adjusted. The optimal damping value is small and depends strongly on model size. Damping hurts full models but improves accuracy on truncated models. We found no interaction between damping and neighborhood size for reasonable neighborhood sizes; ML1M had a small interaction at $n = 10$, but fitting the neighborhood size before the damping term removes this interaction.

These results are consistent with our user-user results in section 5.3 that significance weighting, while necessary for Pearson correlation, does not help cosine similarity. They also suggest that the benefit of damping or significance weighting is in neighborhood selection, not final score computation: by preferring to keep high-confidence neighbors (since low-confidence similarities are damped out), the model is able to achieve higher accuracy; if enough neighbors are available, however, damping does not improve the ability to select neighbors for doing the actual scoring. Model truncation may be able gain benefit by incorporating confidence into the neighbor selection strategy and forgoing explicit damping of similarities.

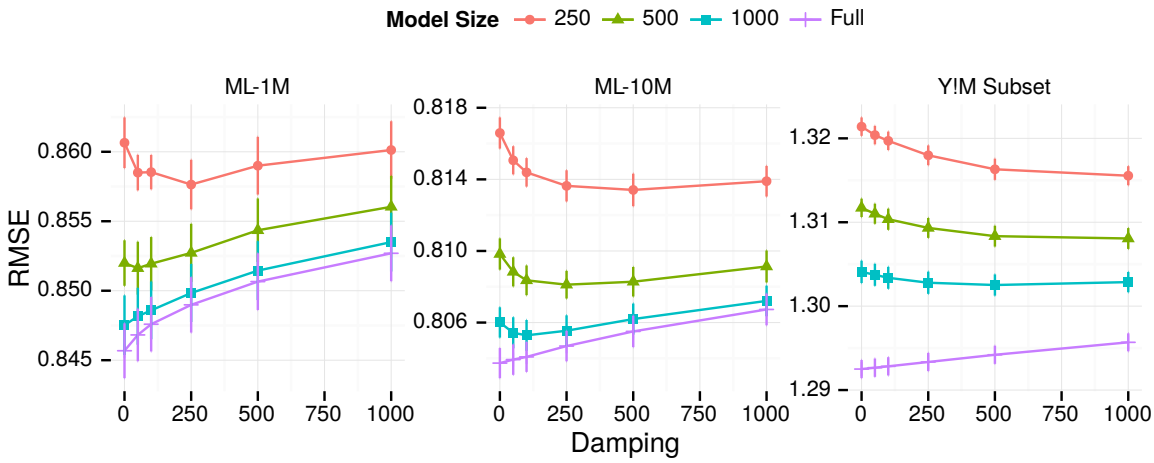


Figure 5.7: Item-item accuracy by similarity damping.

Figure 5.8 shows the impact of the data normalization on recommender accuracy (all recommenders using full models, and use item mean to supply predictions for unscoreable items even when another baseline is used for normalization). Each baseline scorer is used as a normalizer, normalizing rating data by subtracting that baseline's scores prior to computing similarities and rating predictions. We observe two key things here. First, consistent with fig. 5.5, normalizing ratings by the item mean outperforms the user mean that has historically been used. Second, the item-item recommender's performance is not rank-consistent with independent baseline performance. That is, the best-performing baseline, when used as a normalizer, does not necessarily produce the best-performing collaborative filter.

Training Strategy

We propose the following strategy for tuning the configuration of an item-item collaborative filter:

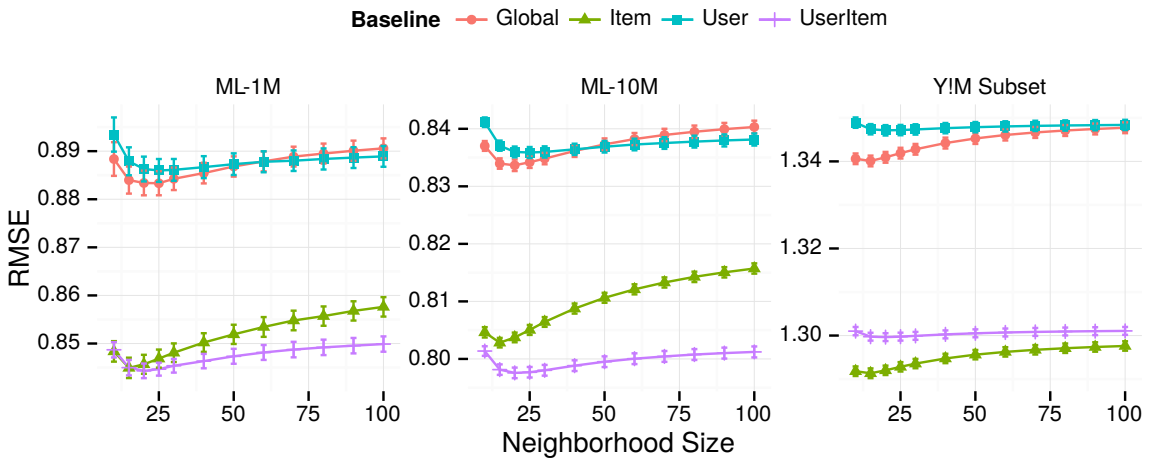


Figure 5.8: Normalizer impact on item-item performance.

1. Use item-user mean as the fallback for unpredictable items.[2]

2. Start with item mean normalization (or item-user; in LensKit, item mean is less computationally expensive, so it is least expensive to start with it)

3. With a full model, start with a small neighborhood size (e.g. 10) and increase until a local minimum is found.

4. Decrease model size for desired size/quality tradeoff.

5. Try the other of item mean and item-user mean normalization to see if there is improvement.

6. If desired, add a small amount of similarity damping to recover lost quality due to model truncation.

[2]Our experiments did not do this due to an experimentation error, but item-item has high enough coverage that any impact on our results should be negligible.

Since model size does not affect the best neighborhood size, steps 2 and 3 can be reversed; in that case, a reasonable neighborhood size (e.g. 20) can be used to pick the desired model size, and then the neighborhood size refined.

The performance difference between using the item mean and item-user mean baselines for normalization seems to vary by data set, with sparsity being a possible reason. More study is needed on a wider array of data sets to understand this relationship more exactly, but using item mean seems to work well.

5.5 Regularized SVD

Figure 5.9 shows the performance of LensKit's gradient regularized SVD (section 3.7.6)implementation on both the 100K and 1M data sets for varying latent feature counts k. λ is the learning rate; $\lambda = 0.001$ was documented by Simon Funk as providing good performance on the Netflix data set [Fun06], but we found it necessary to increase it for the much smaller ML-100K set.

Each feature was trained for 100 iterations, and the item-user mean baseline with a smoothing factor of 25 was used as the baseline predictor and normalization. We also used Funk's range-clamping optimization, where the prediction is clamped to be in the interval $[1, 5]$ ($[0.5, 5]$ for ML-10M) after each feature's contribution is added.

The performance of matrix factorization recommenders is governed by many hyperparameters. These include:

- feature count k
- learning rate λ
- regularization factor γ

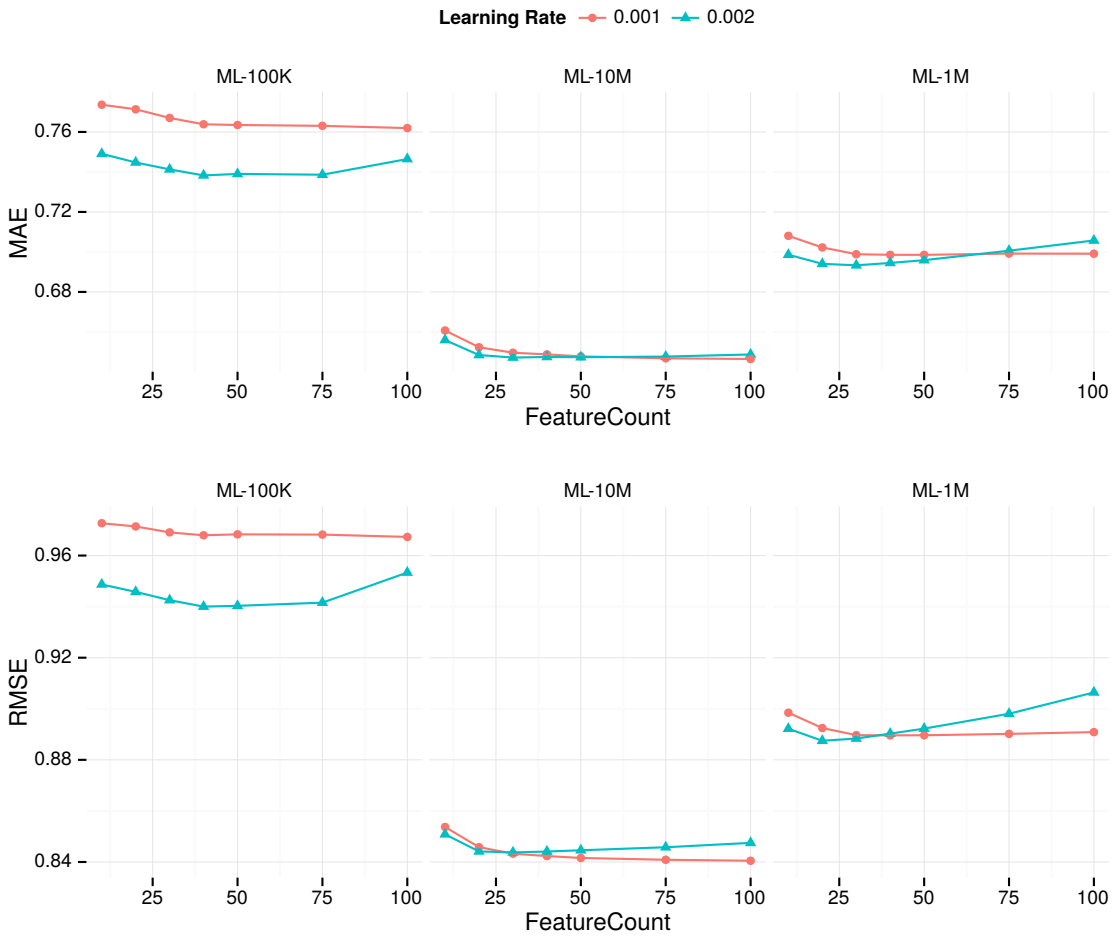


Figure 5.9: Prediction accuracy for regularized SVD.

- per-feature stopping condition (threshold, iteration count, or other criteria)
- baseline predictor

More sophisticated variants have even more parameters. Most of these parameters will affect the final factorized matrix, requiring the model to be retrained for each variant when attempting to optimize them. Optimizing all these parameters by grid search is therefore prohibitively expensive. In practice, a few of the parameters are tuned, such as feature

count, using default values for many of the rest. The optimal values for some of these hyperparameters is also heavily dependent on the data set: the learning control parameters (learning rate and stopping condition) depend greatly, in our experience, on the number of ratings in the data set.

Further, many of the parameters interact. Learning rate and stopping condition naturally interact — a higher learning rate will accelerate convergence, though at the likely expense of accuracy. Our experiments have also found the regularization term and feature count to interact with the stopping condition in minimizing the recommender's error.

To decrease the search space, we have attempted to find more automatic strategies for determining when to stop training. The process of learning an SVD needs two stopping conditions: it needs to know when to stop training each feature, and when to stop training new features.

If we can determine when to stop either (or both) of these two processes in a parameter-free fashion (or based on parameters whose values are unlikely to be dataset-dependent), then we can decrease the dimensionality of the hyperparameter search space and make tuning significantly more efficient. A similar approach may also be applicable to other parameters, but we focus here on the stopping condition.

5.5.1 Training a Feature

Any method for determining when to stop training a feature can depend only on information available during the training process. The information available while training a feature includes:

- The number of epochs computed so far
- The training error for each epoch

- If the training algorithm reserves a set of ratings for tuning/validation, the error on these ratings after each epoch
- The average estimated gradient in an epoch (and its magnitude)
- Derivatives of any of these values

Directly thresholding training error is impractical, because the achievable error will differ between data sets, rating ranges, etc. Applying a threshold to the change in training error between two epochs (thresholding the derivative of training error) is a feasible solution, however: if the change is small, especially over multiple iterations, then the feature values have likely converged. Similarly, the change in validation RMSE can be thresholded. Thresholding the magnitude (L_2 norm) of the average estimated gradient (change in user and item feature weight vectors in an epoch) is also practical, with a low magnitude indicating convergence. We have not yet tested any second derivatives of these features.

The learning rate is also key in the process of training a feature. So far, we have only tested fixed learning rates. It may be that dynamic learning rate schedules would improve the performance, either in training time or output quality, generally, and that it may make thresholding approaches more useful.

5.5.2 Training New Features

Typically, the number of features is fixed in advance, and the initial value (rather than 0) is assumed for the user/item values for features not yet trained; this has the unfortunate side effect of making the training for each feature dependent on the number of features not yet trained. Nonetheless, we have tested approaches that relax this, training each feature independent of the number of remaining untrained features and attempting to automatically detect whether to continue.

The data available to decide whether to train another feature include:

- The number of features
- The training error of the last pass for each feature
- The error on a tuning/validation set of ratings after each feature
- The weight of the feature (product of the L_2 norms of its user and item vectors; this is the singular value in a true SVD)
- Derivatives of any of these values

These are subject to similar considerations as the training stopping criteria. Thresholding the difference in feature weights is similar to using skree plots to pick the number of latent factors in factor analysis.

5.5.3 Tuning Results

Unfortunately, none of these strategies can reliably match or beat well-selected parameter values on ML-1M: 25–30 features for 125–50 epochs per feature. If they cannot reliably find known good values on a well-understood data set, we are hesitant to trust them for tuning on previously-unseen data.

We have yet to find a good way to disentangle stopping training on either an individual feature or the entire model. FunkSVD accuracy seems to be fairly stable in the face of reasonable values; differing slightly from our to-beat values does not produce large differences in RMSE or nDCG. However, being unable to reliably match or beat the performance of these values using more automated techniques hurts our ability to develop a tuning strategy. A viable strategy will need to have more sophistication than the first-order approaches we have listed here.

5.6 Impact of Rank-Based Evaluations

Many approaches to recommendation interpret ratings as numbers on a linear scale.[3] That is, they assume that a 5-star movie is as much better than a 4-star movie as a 4-star is better than a 3-star movie. This assumption is widely known to be incorrect: ratings provide a partial ordering, but they are not a measurement. When a user rates one item 4 stars and another 5, the system can take that as evidence that they prefer the second to the first, but cannot directly infer *how much* more the user likes the second. Ratings are effectively Likert-style feedback [Bla03], but our algorithms are interpreting it as statements of absolute preference values. The problems with this assumption have been gaining increasing attention in the research community; notably, new recommendation techniques have been developed to get away from this problem, such as predicting ratings with ordinal logistic regressions [KS11].

These assumptions are not just made by the algorithms themselves. Many common prediction accuracy metrics, notably MAE and RMSE, also assume that subtracting ratings is a meaningful thing to do. Amatriain [Ama11] proposed that this is causing problems for recommender research: since our metrics make such a false assumption about the data over which they are operating, we may well be incorrectly optimizing and measuring the recommenders themselves.

We seek to understand whether this flaw in recommender design and evaluation corresponds to decreased effectiveness of recommender algorithms. Even if most algorithms are based on a flawed premise — that user ratings provide an absolute measurement of preference — it may be that these algorithms are still sufficiently effective.

Since LensKit allows us to easily test many different recommenders and configurations with different evaluation metrics, we can hopefully provide some data to inform this discus-

[3] This work was done in collaboration with Michael Ludwig, John T. Riedl, and Joseph A. Konstan and much of it was published in [Eks+11].

sion. We therefore tested a selection of recommenders with both nDCG — a rank-accuracy metric — and RMSE. If the relative performance of the algorithms differed, that would be evidence that using distance-based accuracy metrics is indeed leading us astray.

Figure 5.10 shows some of the permutations of user-user and item-item collaborative filtering tested in sections 5.3 and 5.4 using nDCG. In this setup, we compute nDCG only over the test items, ranking them by prediction and using the user's rating as the gain for each item. This converts nDCG from its traditional top-N evaluation usage into a discounted rank-accuracy metric. There is little difference in the relative performance of these the variants measured under nDCG and under MAE or RMSE (figs. 5.3 and 5.4). Of particular note is the fact that Spearman correlation — a rank-based approach to computing user similarity — continues to perform noticeably worse than distance-based methods. We might expect it to perform better when using a rank-based evaluation metric.

This lack of change as a result of using nDCG does not mean that there is no impact on recommender effectiveness as a result of distance-based evaluation. It may be that our current families of algorithms cannot easily be adjusted to think of user preference in terms of ranks and entirely new approaches are needed. It could also be the case that more sophisticated experimental frameworks, particularly user studies or field trials, are necessary to see an actual difference. The better-performing algorithms in our experiment achieve over 0.95 nDCG, putting them within 5% of being perfect within the measurement capabilities of the metric. Achieving the remaining 5% may not be feasible with the noise inherent in user ratings (as it is likely that ratings are not entirely rank-consistent with user preferences), and may not accurately measure real user-perceptible benefit.

If we go further from rank evaluation, however, and start considering metrics in top-N configurations, optimal tuning values start to change more. Figure 5.11 shows the top-N nDCG of various item-item CF configurations. Rather than ranking the test items, as

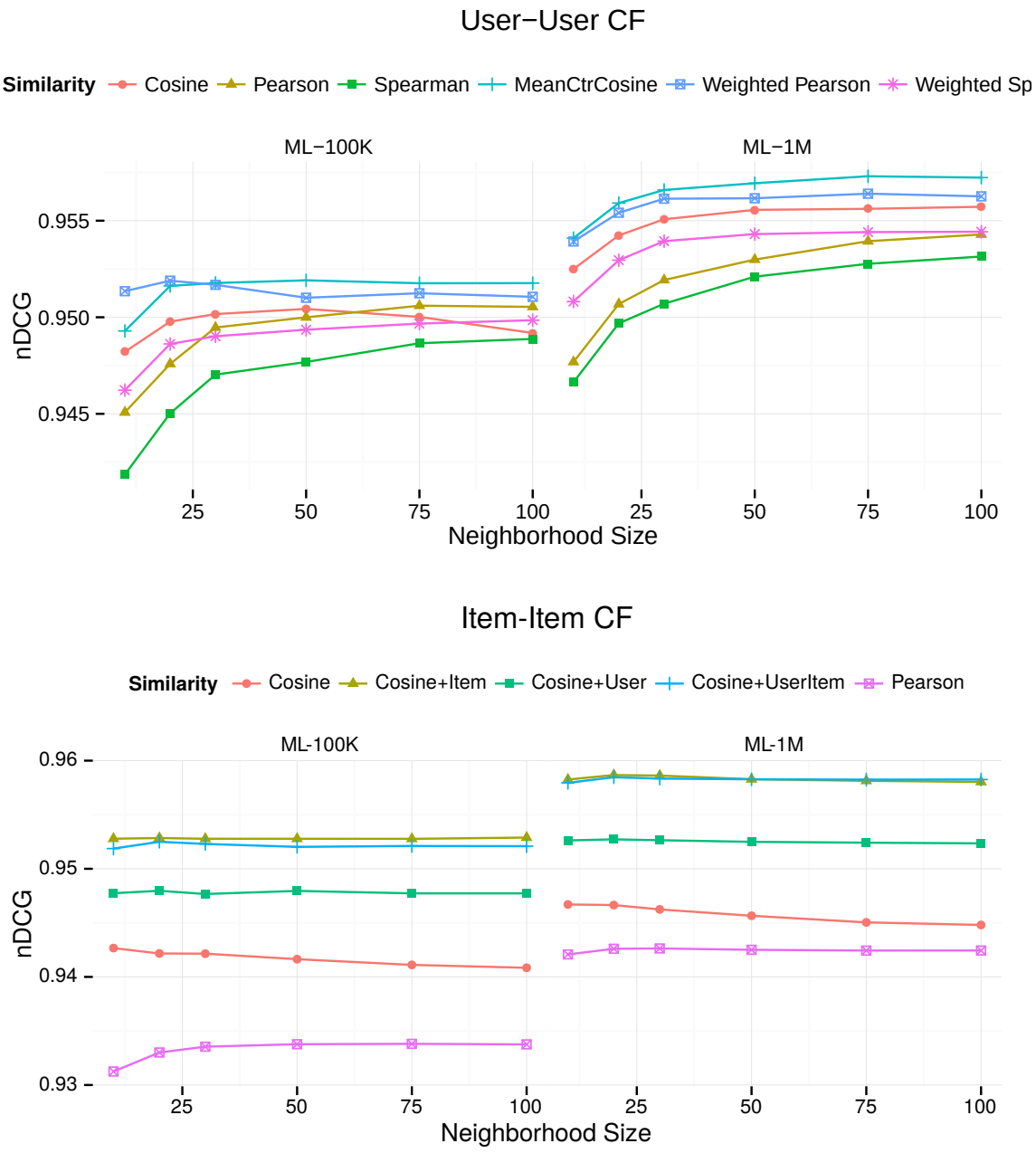


Figure 5.10: Rank-based evaluation of CF configurations.

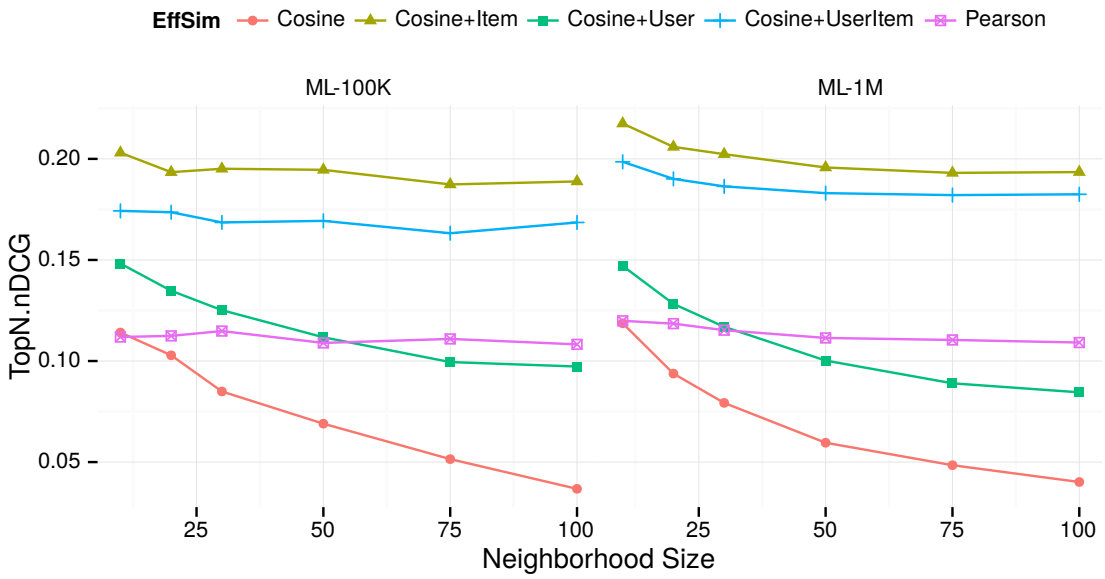


Figure 5.11: Top-*N* evaluation of CF configurations.

was done to compute nDCG in fig. 5.10, each recommender in this setup generated a recommendation list from a set of candidate items containing the user's test items plus 100 randomly-selected decoy items. Here, the best similarity function remains the same, but the optimal choice of neighborhood size changes considerably.

Mean Reciprocal Rank (MRR) also produces some striking differences from RMSE. Figure 5.12 shows the performance of item-item with different normalizers, like fig. 5.8 but with MRR. MRR was computed like top-*N* nDCG: the recommender produced recommendations from a candidate set containing the user's rated items plus 100 random items, and we considered an item 'relevant' if the user had rated it at least 3.5 out of 5 stars. The results for the MovieLens data sets are consistent with RMSE (except for neighborhood size choice for non-optimal normalizer), but the optimal choice of normalizer for Y!M is very different. From this we conclude that tuning a recommender for performance on a top-*N* metric

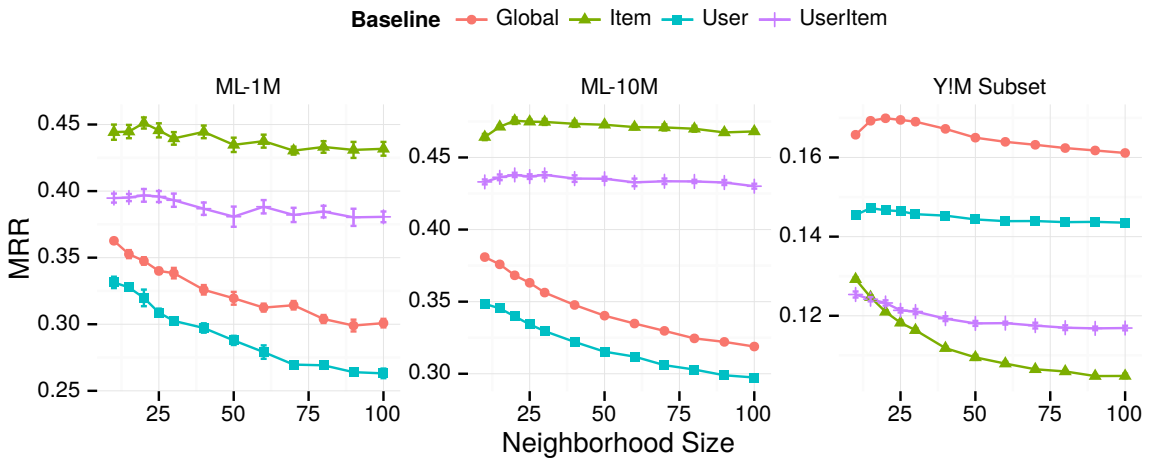


Figure 5.12: Item-item top-*N* performance by baseline normalizer

may result in a substantially different configuration — and very different recommender — than tuning for a prediction accuracy metric. This raises the question of whether top-*N* and prediction accuracy work on the same algorithm are really testing the same recommender. However, the difficulties inherent in top-*N* evaluation of recommender systems keep us from inferring too much from them. This could mean that achieving good recommendation performance requires a different tuning than for prediction performance, or if MRR is a poor metric for tuning a recommender. More work is needed to understand the mapping between algorithms and user experience.

We still prefer to optimize for prediction accuracy or rank accuracy, as top-*N* performance is subtle, dependent on the parameters of the evaluation (such as the set of candidate items), and fraught with uncertainty in the face of missing data. However, it does show that the choice of metric, particularly changing the evaluation setup away from prediction accuracy, can have a significant impact on the optimal choice of algorithm configuration.

Chapter 6

When Different Algorithms Fail

In this section, we present an offline experiment designed to elucidate one way in which different recommender algorithms may differ: the users and items for which they make errors.[1] We are particularly interested in *mispredictions*, and want to know how often and when one algorithm erroneously predicts a user's rating for an item but another algorithm correctly makes that prediction. We consider a prediction to be correct if it is within $1/2$ star on a 5-star scale (the granularity of MovieLens's rating input).

Prediction accuracy has traditionally been assessed by measuring the aggregate error, using either MAE or RMSE or a related metric. This has two significant problems. First, since ratings are ordinal, the idea of measuring error against them is problematic, as discussed in section 5.6. Second, they don't seem to measure well the user experience of rating prediction: a user is likely to notice if a rating prediction is off by a half-star or more, but is unlikely to notice small differences in errors (e.g. being 0.25 vs. 0.28 off).

Examining mispredictions allows us to see how often an algorithm makes a prediction that is close enough to be reasonable, or how often its predictions will be far enough off to make a visible difference when displayed to the user. It also gives us a binary measure by which we can say one algorithm was wrong but another was right.

In this experiment, we address three research questions:

[1] The bulk of this section has been published in [ER12]

RQ1 How do classic collaborative filtering algorithms differ in their ability to correctly predict ratings within a fixed tolerance? How does this relate to their performance on traditional prediction error measures?

RQ2 Do different algorithms make different prediction errors?

RQ3 Are different algorithms better for different users?

We are also interested in whether different algorithms are better for different items, but so far have not made much progress on that front.

These questions have implications for selecting algorithms and for combining algorithms into hybrids [Bur02]. If two algorithms make roughly the same errors, then there may not be much benefit to combining them, at least with simple hybridization techniques; the additional marginal signal is likely not worth the computational overhead. Two algorithms that make very different errors seem likely to be drawing on and contributing different signals to the final recommender system. Also, in selecting a single algorithm, an application may prefer to pick an algorithm that is close more often rather than an algorithm with lower aggregate error, if there is a difference.

The success of hybrid approaches such as feature-weighted linear stacking (FWLS) [Sil+09], that adapt a hybrid based on properties of the user and item being recommended, suggest that the answers to RQs 2 and 3 are in the affirmative, but we seek to demonstrate this more concretely and work towards a more transparent model for selecting and combining recommenders. We are not aware of much published research that takes apart the hybrids learned by FWLS and related techniques to attempt to understand what the constituent algorithms' strengths and weaknesses are, and the value contributed by each.

6.1 Methodology

This experiment uses the ML-10M data set with 5-fold cross-validation, again using LensKit's default user-based strategy. For each user, we held out 20% of their ratings as test ratings for the recommender.

We then used LensKit to train and run five recommender algorithms on the data, outputting the predictions for each test rating for analysis. We used the following algorithms, choosing parameters based on prior results in the research literature and experience tuning LensKit for the MovieLens data sets in the previous sections:[2]

- Item-user mean, the item's average rating plus the user's mean offset with a Bayesian damping term of 25. This algorithm was also the baseline for all others — if they could not make a prediction, the item-user mean was used.
- Item-item collaborative filtering with a neighborhood size of 30 and ratings normalized by subtracting the item-user mean.
- User-user collaborative filtering with a neighborhood size of 30, using cosine similarity over user-mean-normalized ratings. In the predict stage, ratings were normalized by z-score [HKR02; Eks+11].
- FunkSVD with 30 features and 100 training iterations per feature.
- Apache Lucene [Apa11] as a tag-based recommender. Since the ML-10M data set contains tags for movies, we created a document for each movie containing its title, genres, and tags (repeating each tag as many times as it was applied). Recommenda-

[2]We have gained more experience in tuning and found better parameter values since this experiment was originally conducted. For consistency with the published work, we have retained the original parameter values.

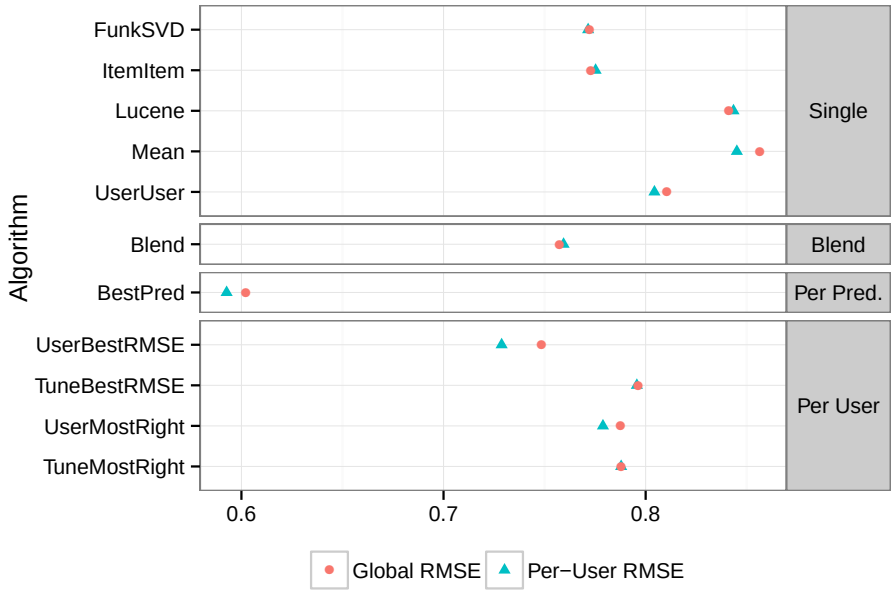


Figure 6.1: Algorithm prediction error.

tion were then computed as in item-item collaborative filtering, with item neighborhoods and scores computed by a Lucene `MoreLikeThis` query.

After running the recommenders, we processed each test set to discard all users with fewer than 10 test ratings (ultimately using 44,614 of the 69,878 users in ML-10M) We then split and each retained user's test ratings into two sets: 5 ratings from each user went into a tuning set, and the remaining ratings stayed in the final test set.

6.2 Basic Algorithm Performance

Figure 6.1 shows the overall RMSE achieved by each of the recommender algorithms (the *Single* section), and fig. 6.2 shows the fraction of predictions each algorithm got correct. The *Blend* algorithm is a linear combination of all 5 algorithms, trained on the tuning set.

The *BestPred* algorithm is an oracle switching hybrid: for each user-item pair, it uses

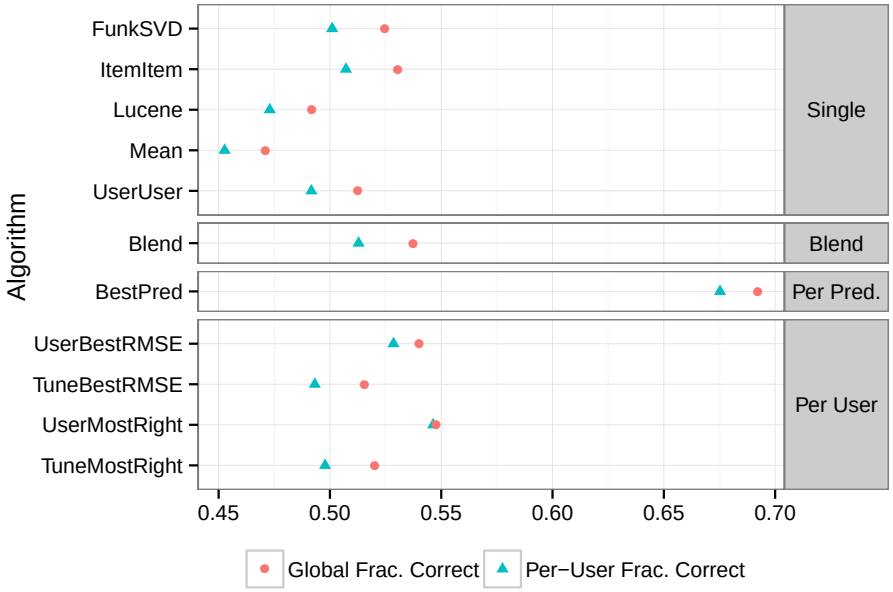
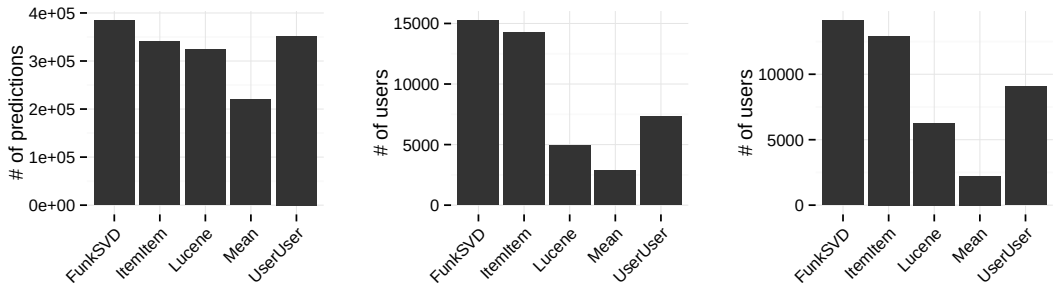


Figure 6.2: Correct predictions by algorithms.



(a) By prediction. (b) By user RMSE. (c) By user correct predictions.

Figure 6.3: Distribution of best algorithms, by prediction and user.

the algorithm whose prediction is closest to the user's true rating. This places a lower bound on the accuracy achievable by any switching hybrid strategy on this data set. Figure 6.3(a) shows how often this hybrid chose each algorithm.

The four per-user algorithms are switching hybrids that operate on a user-by-user basis. *UserBestRMSE* and *UserMostRight* are oracle hybrids, picking the algorithm that achieves the lowest RMSE or highest fraction of correct predictions, respectively, for each user; figs. 6.3(b) and 6.3(c) show how often these hybrids picked each algorithm. The *Tune* variants are realistic switching hybrids that use tuning set performance to select the algorithm to use on each user's test ratings.

	MAE	RMSE	FracCorrect
MAE	1.00		
RMSE	0.97	1.00	
FracCorrect	-0.50	-0.41	1.00

Table 6.1: Error metric correlation matrix

Table 6.1 shows the correlation of the Fraction Correct method with the MAE and RMSE, by user. Fraction Correct is correlated with prediction error, but not strongly: the correlation with RMSE is less than 0.5. Figure 6.4 shows how often the two measures agree on which algorithm is best; they only agree 32.1% of the time.

The RMSE and correct-prediction metrics for evaluating algorithms are mostly rank-consistent, however: an algorithm with lower RMSE generally gets more predictions correct. This is even the case with the per-user tuning-based hybrids: picking the algorithm that has the most correct predictions over the user's tuning ratings results in a hybrid that does slightly better on both RMSE and fraction correct than using the tuning RMSE for selection.

There are two notable exceptions to this. First, the user oracle hybrid does not have very

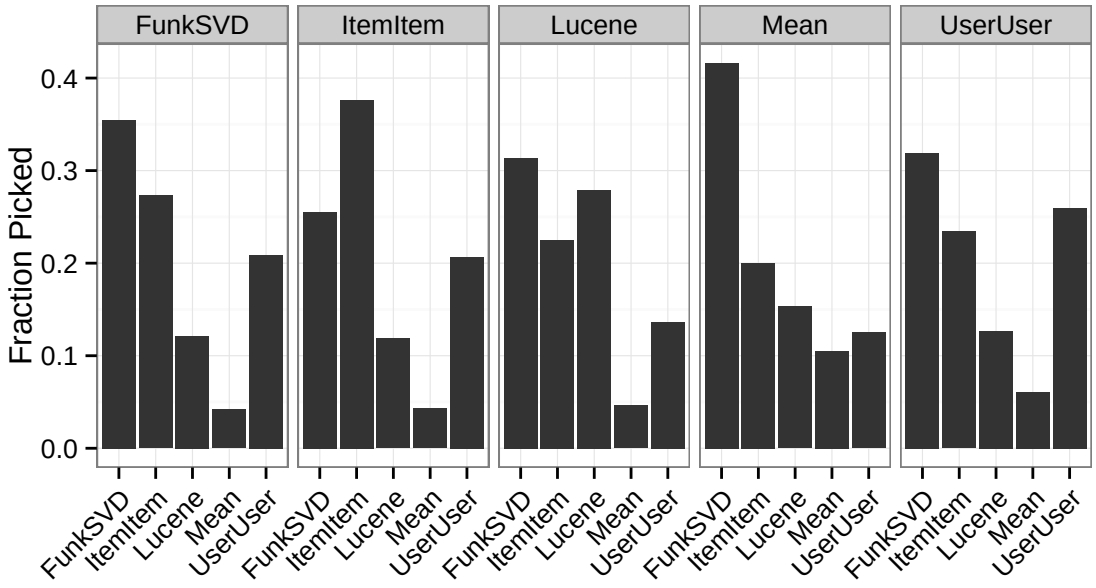


Figure 6.4: Comparative distributions of picked algorithms by user. Each panel contains the users for which the RMSE oracle hybrid picked the given algorithm, and shows the distribution of algorithms picked by the Most Right oracle hybrid.

low RMSE compared to the other user-based algorithms, while it produces the most correct predictions. Second, the best single algorithms — FunkSVD and ItemItem — are essentially tied on RMSE (FunkSVD has a nearly immeasurably small advantage, but our experiments with these algorithms usually have a consistent, small advantage for FunkSVD), but item-item gets more predictions correct. This suggests that when tuning a recommender for predicting user ratings, optimizing for making predictions correctly may result in different design decisions than optimizing RMSE or other metrics.

On both metrics, item-item and FunkSVD are clear winners among the single algorithms, with user-user coming in behind them. Thus, our answer to RQ1 is that item-item produces the most correct predictions, and that mispredictions and RMSE are broadly correlated but differ in some key places, including comparing the performance of the best-performing algorithms.

Primary	# Correct	Secondary				
		ItemItem	FunkSVD	UserUser	Lucene	Mean
ItemItem	859,600		120,078 (15.8%)	**131,356 (17.3%)**	126,154 (16.6%)	117,320 (15.4%)
FunkSVD	850,139	129,539 (16.8%)		144,986 (18.8%)	**153,198 (19.9%)**	129,842 (16.9%)
UserUser	830,563	160,393 (20.3%)	**164,562 (20.8%)**		157,802 (20.0%)	118,440 (15.0%)
Lucene	797,193	188,561 (22.9%)	**206,144 (25.0%)**	191,172 (23.2%)		110,492 (13.4%)
Mean	763,395	213,525 (24.9%)	**216,586 (25.3%)**	185,608 (21.7%)	144,290 (16.8%)	

Table 6.2: Correct predictions by primary and secondary algorithm.

Algorithm	# Correct	% Correct	Cum. % Correct
ItemItem	859,600	53.0	53.0
UserUser	131,356	8.1	61.1
Lucene	69,375	4.3	65.4
FunkSVD	44,960	2.8	68.2
Mean	16,470	1.0	69.2
Unclaimed	498,850	30.8	100.0

Table 6.3: Cumulative correct predictions by algorithm.

6.3 Relative Errors

To answer RQ2, we want to know when one algorithm misses a prediction but another gets it correct. Table 6.2 shows, for each algorithm ('Primary'), the number of predictions it correctly made, followed by the number of its errors that each other algorithm could correctly predict. The most correct algorithm — ItemItem — can still have 17.3% of its errors (8.1% of the total predictions) picked up by a single additional algorithm. Table 6.3 shows the cumulative good predictions for the 5 algorithms. This table is computed by first picking the algorithm that has the most good predictions. The remaining algorithms

are selected and computed by picking the algorithm which has the most good predictions that no prior algorithm has correctly made and adding it to the table. While ItemItem only correctly predicts 53.1%, the algorithms together can predict 69.3% of the predictions.

This result provides an initial affirmative answer to RQ2: algorithms differ in which predictions they get right or wrong. It is also robust to higher thresholds; using a threshold of 1.0 stars for good prediction scales the ItemItem hit count up and the other hit counts correspondingly down, but does not change the relative ordering of algorithms.

The ability of the oracle hybrids to outperform single algorithms — even when the hybrid selects algorithms on a per-user basis — provides further evidence for useful differences in the errors made by different algorithms.

When selecting algorithms to deploy in an ensemble recommender, it is not necessarily desirable just to pick the ones that perform the best. If two algorithms are highly correlated in the errors they make, failing in the same cases, then including both of them will likely not provide much benefit. In selecting algorithms, we look for the following criteria:

- Unique benefit — individual algorithms should contribute unique benefit with respect to the other algorithms in the ensemble.
- Distinguishability — it should be possible to figure out how to blend the algorithms or to select which one to use.
- Tractability — given two algorithms with similar benefit, prefer algorithms that are less expensive to operate.

In general, we found all algorithms to be highly correlated, as shown in the correlation matrix in table 6.4. FunkSVD and ItemItem had the highest correlation. Also, if either of them is used as the primary algorithm, the other is not the best secondary algorithm,

	FunkSVD	ItemItem	Lucene	Mean	UserUser
FunkSVD	1.00				
ItemItem	0.95	1.00			
Lucene	0.89	0.91	1.00		
Mean	0.90	0.92	0.93	1.00	
UserUser	0.93	0.93	0.90	0.94	1.00

Table 6.4: Correlation of algorithm errors.

Algorithm	# Correct	% Correct	Cum. % Correct
FunkSVD	850,139	52.5	52.5
Lucene	153,198	9.5	61.9
UserUser	69,331	4.3	66.2
ItemItem	32,623	2.0	68.2
Mean	16,470	1.0	69.2
Unclaimed	498,850	30.8	100.0

(a) SVD first

Algorithm	# Correct	% Correct	Cum. % Correct
FunkSVD	850,139	52.5	52.5
UserUser	144,986	8.9	61.4
Lucene	77,543	4.8	66.2
ItemItem	32,623	2.0	68.2
Mean	16,470	1.0	69.2
Unclaimed	498,850	30.8	100.0

(b) SVD followed by user-user

Table 6.5: Cumulative correct predictions by algorithm, alternate permutations.

as can be seen in table 6.2; further, generating table 6.3 with FunkSVD first reverses its position with item-item, putting item-item last among the collaborative filtering algorithms (see table 6.5(a)). This suggests that ItemItem and FunkSVD tend to make many of the same mistakes, so using both of them together may not be as useful as combining one of them with an algorithm that provides substantially more marginal benefit.

The alternate permutations in table 6.5 also show that the greedy approach in table 6.3 does not produce an optimal two-algorithm hybrid: an oracle hybrid of FunkSVD with

Lucene would achieve a correct prediction rate of 61.9%, and combining FunkSVD with UserUser beats the ItemItem/UserUser combination.

Algorithms do differ in the errors that they make, and the marginal benefit of each over the others is subtle. For the ML-10M data set, ItemItem is the best single algorithm, but optimal oracle switching hybrids of 2-3 algorithms would not include it.

6.4 Comparing by User

The performance of the per-user oracle hybrids — and the diversity of algorithms selected, as seen in figs. 6.3(b) and 6.3(c) — indicate that, at a high level, the answer to RQ3 is also 'yes'. Different algorithms do perform better — lower error, more correct predictions — for different users.

We now want to see if we can distinguish between users for which different algorithms perform better. To simplify the problem, we consider the question of whether ItemItem will outperform UserUser for a particular user. Building a model that can successfully predict the best algorithm for a user may result in a more useful hybrid than using the tuning set to pick the best algorithm (the pick-by-tuning approach seems likely to overfit the user's tuning ratings; modeling over the tuning of many users would hopefully avoid this problem).

Table 6.6 shows two logistic regressions attempting to predict whether item-item will outperform user-user for each user, using both 'best RMSE' and 'most correct' as algorithm selection metrics. When using RMSE to select an algorithm, item-item tends to outperform user-user for users with many ratings or high variance in their ratings. When using Most Right, however, item-item does worse for users with many ratings, as well as for users with a high average rating. When learning both of these models, we held out 20% of the users as a test set; fig. 6.5 shows the ROC curves of these two models. They do noticeably better

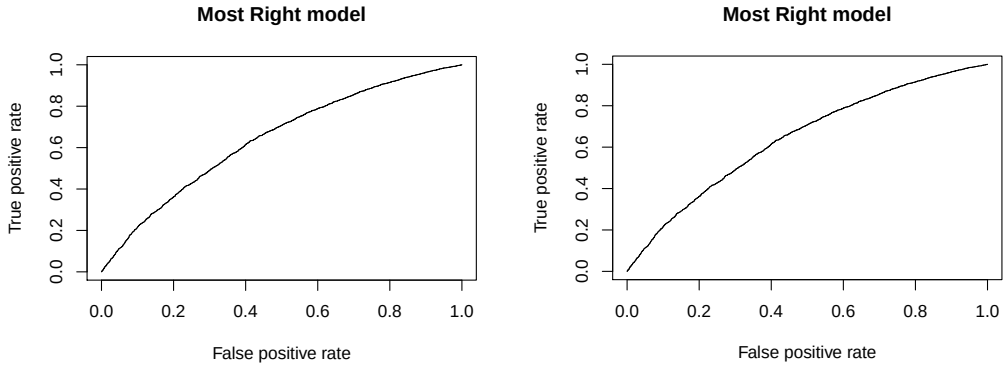
	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-2.6008	0.0811	-32.08	0.0000
LogCount	1.5391	0.0378	40.73	0.0000
RatingVar	0.2190	0.0270	8.11	0.0000

(a) By RMSE

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	7.2843	0.1500	48.55	0.0000
LogCount	-1.3023	0.0346	-37.60	0.0000
MeanRating	-1.0651	0.0307	-34.66	0.0000

(b) By Most Right

Table 6.6: Logistic regressions predicting that item-item outperforms user-user.



(a) RMSE model (AUC 0.637)

(b) Most Right model (AUC 0.643)

Figure 6.5: ROC of user-user vs. item-item models from table 6.6

than guessing at picking which of the algorithms will do better.

From these analyses, we can see three things:

- Different algorithms do, in fact, do better or worse than others for different users (answering RQ3 in the affirmative).
- Certain features of the user's profile can predict with modest success whether one algorithm will outperform another for that user.

- Traditional error metrics and counting correct predictions do not agree on which algorithm is better for a particular user, while they do agree on the relative performance of algorithms overall.

6.5 Conclusion

In this experiment, we have investigated the errors (mispredictions) made by different recommender algorithms. We have found that different algorithms make different errors — where one algorithm misses a prediction, another algorithm may make that prediction correctly. And we have demonstrated that different algorithms perform better or worse for different users.

We have also found significant overlap in the errors made by all recommenders we tested, but particularly item-item CF and FunkSVD. Our data suggest that hybrid recommenders with limited resources would do better to combine one of these algorithms with a significantly different algorithm, rather than combining the two of them.

More work is needed to understand better why the algorithms are performing differently on different users. For what users is item-item, FunkSVD, or user-user a superior recommender algorithm? We have some results on this question, but they are heavily dependent on the choice of metrics for identifying the 'best' algorithm for a particular user. Given these two metrics that exhibit significant differences in their assessments of algorithms — and the choices they yield on a user-by-user basis — we need further study to understand the implications of measuring recommender error vs. correct predictions in order to know how best to assess an algorithm's suitability for a user.

User studies and qualitative investigation of the items and users themselves will likely be helpful in further elucidating the specific behavior of each algorithm. So far, our work

has focused only on generic statistics of users and items in the rating set; seeing what actual items are being mispredicted and collecting user feedback on erroneous predictions or bad recommendations would hopefully provide further insight into how the algorithms behave.

Chapter 7

User Perception of Recommender Differences

HAVING DEVELOPED an extensive suite of tools for recommender experimentation and applied them in two offline contexts, we now complete the arc by investigating how the differences between the recommendation solutions we have considered affect the users of the recommender system. If we are going to engineer nuanced recommender solutions to a diverse set of user information needs, we need to understand how various recommendation options differ in ways that relate to their ability to meet users' needs.

To that end, we now present a user study in which we asked users of the MovieLens movie recommendation service to provide comparative judgements of the differences they see in the outputs from common recommender algorithms.[1] In the time since we first published LensKit, MovieLens has been updated to use LensKit to provide its recommendations, allowing us to easily plug different algorithms into MovieLens and see how they perform. At the time of writing, we are also preparing the general release of a new version of the MovieLens platform, providing an opportunity to conduct an experiment in a context where the question of user preference among recommender algorithms has real meaning.

The experiment described in this chapter is intended to answer the following questions:

RQ1 How are users' overall preferences for recommendation lists predicted by the subjective properties of those lists?

[1] This work was done in collaboration with F. Maxwell Harper, Martijn C. Willemsen, and Joseph A. Konstan. It has been accepted for publication in RecSys 2014 [Eks+14].

RQ2 What differences do users perceive between the lists of recommendations produced by commonly-used collaborative filtering algorithms?

RQ3 How do objective algorithm performance metrics relate to users' subjective perception of recommender outputs?

This experiment asks users to directly compare recommendation lists produced by popular recommender algorithms. We specifically explore item-item, user-user, and SVD algorithms, looking at user perceptions of accuracy, personalization, diversity, novelty, and overall satisfaction. Each user provided a first-impression preference between a pair of algorithms, subjective comparisons of the algorithms' output on our dimensions of interest, and selected an algorithm for future use. We build a model that predicts both initial user preference and eventual user choice after more in-depth reflection on the recommendations using the subjective perceptions and objective measures of the recommender algorithms and their output.

While this experiment focuses on one application — general-purpose movie recommendation — that is admittedly well-studied, it uncovers subjective characteristics of recommender behavior that explain users' selections in a manner that provides a good basis for generalization, replication, and further validation. We report specific relationships that can be tested for validity in additional contexts, providing much greater insight into what aspects of algorithm suitability for movie recommendation are task-specific and what are more general behaviors.

In addition to answering our immediate questions, the data collected in this survey should be a useful ground truth for calibrating new offline measures of recommender behavior to more accurately estimate how algorithms will be experienced by their users. We use it perform some of this analysis in section 7.2.4.

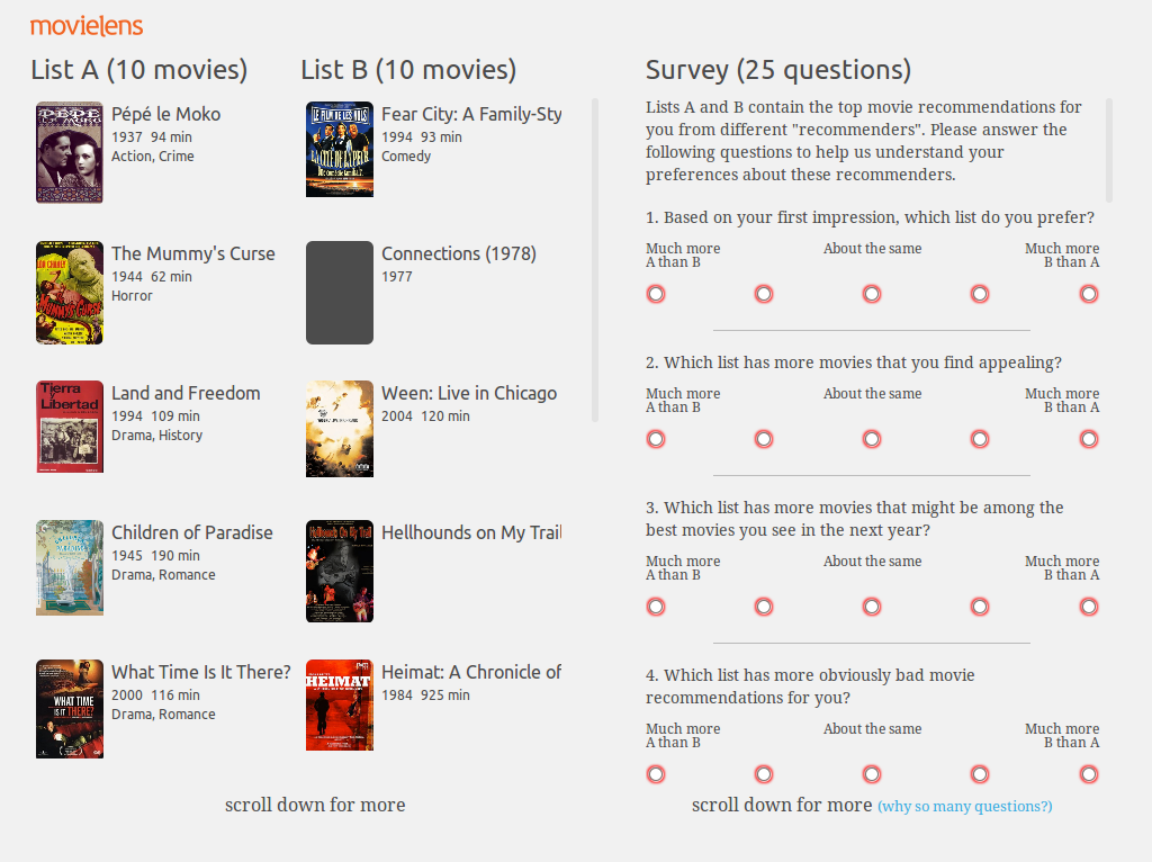


Figure 7.1: Screen shot of the experiment interface. Clicking on a movie in the list opens a pop-over with additional movie details.

7.1 Experiment Design

To assess the differences among various algorithms for recommending movies with explicit user ratings, we conducted an experiment in which users reviewed two lists of recommendations and took a survey comparing them. Figure 7.1 shows a screenshot of the experimental interface.

```
// we have a 5-star scale with half stars
domain minimum: 0.5, maximum: 5.0, precision: 0.5

// use user-item personalized mean as the baseline predictor
bind (BaselineScorer, ItemScorer) to UserMeanItemScorer
bind (UserMeanBaseline, ItemScorer) to ItemMeanRatingItemScorer

// just a little mean damping seems to improve things
set MeanDamping to 5
```

Listing 7.1: Common configuration for recommender algorithms.

7.1.1 Users and Context

We conducted our experiment on users of MovieLens, a movie recommendation service. The survey was integrated into a beta launch of a new version of MovieLens; we invited active users to preview the beta with an on-site banner and required them to participate in the experiment prior to using MovieLens Beta. 1052users attempted the survey, of which 582completed it. Since we limited recruiting to active users, all users had at least 15 ratings (the median rating count was 473).

7.1.2 Algorithms

For this experiment, we tested three widely-used collaborative filtering algorithms as implemented in LensKit version 2.1-M2 [Eks+11]. To tune the algorithm parameters, we used the item-item CF configuration in the MovieLens production environment and values reported in the published literature [Eks+11; Fun06] as a starting point and refined the configurations with 5-fold cross-validation over the MovieLens database (using RMSE and prediction nDCG as our metrics to optimize) and manual inspection of recommender output. This resulted in the following algorithm configurations:

```
// use item-item CF
bind ItemScorer to ItemItemScorer

// use cosine
bind VectorSimilarity to CosineVectorSimilarity

// we'll normalize by item mean, so use the item-by-item builder
bind ItemItemBuildContext toProvider ItemwiseBuildContextProvider
within (ItemItemBuildContext) {
    bind VectorNormalizer to MeanCenteringVectorNormalizer
}

// use item mean to normalize ratings when producing scores
within (ItemItemScorer) {
    bind UserVectorNormalizer to BaselineSubtractingUserVectorNormalizer
    bind (BaselineScorer, ItemScorer) to ItemMeanRatingItemScorer
}

// set up sizes and thresholds
set ModelSize to 4000
set NeighborhoodSize to 20

set MinNeighbors to 2
set ThresholdValue to 0.1

// the global item scorer allows us to ask for similar movies
bind GlobalItemScorer to ItemItemGlobalScorer
within (GlobalItemScorer) {
    // MinNeighbors must be 1 to enable "movies like this" on movie details pages
    set MinNeighbors to 1
}
```

Listing 7.2: Item-item algorithm configuration.

```
bind ItemScorer to UserUserItemScorer

bind VectorSimilarity to CosineVectorSimilarity

bind UserVectorNormalizer to BaselineSubtractingUserVectorNormalizer
within (UserVectorNormalizer) {
    bind (BaselineScorer, ItemScorer) to UserMeanItemScorer
    bind (UserMeanBaseline, ItemScorer) to ItemMeanRatingItemScorer
}
bind NeighborFinder to SnapshotNeighborFinder

set NeighborhoodSize to 30

set ThresholdValue to 0.1
set MinNeighbors to 2
```

Listing 7.3: User-user algorithm configuration.

```
bind ItemScorer to FunkSVDItemScorer
// FunkSVD will automatically use the user-item baseline, which is correct

set FeatureCount to 50
set IterationCount to 125
```

Listing 7.4: SVD algorithm configuration.

- Item-item CF [Sar+01] with 20 neighbors, model size of 4000, cosine similarity, item mean centering, neighbor threshold of 0.1, and requiring 2 neighbors to make a prediction (listing 7.2).

- User-user CF [HKR02] with 30 neighbors, cosine vector similarity between users, and normalizing user ratings by subtracting the personalized user-item mean, a neighbor threshold of 0.1, and requiring 2 neighbors to make a prediction; we additionally applied a small Bayesian damping of 5 to the user and item means for normalization

(listing 7.3).

- SVD with the FunkSVD [Fun06; Pat07] training algorithm, using 50 features, 125 training epochs per feature, user-item mean baseline with damping of 5, and the LensKit default learning rate of 0.001 and regularization factor of 0.015 (listing 7.4).

In addition to their specific configurations, each algorithm included a common core configuration (listing 7.1) and additional configuration to connect to the MovieLens database.

For each user, we randomly selected two of the algorithms. For each algorithm, we computed a recommendation list containing the 10 movies with the highest predicted rating among those the user had not rated, sorted by predicted rating. We presented these lists as 'List A' and 'List B' (the ordering of algorithms was randomized).

Most studies that measure such user experiences, employ a between-subjects design in which the users only see one condition (i.e. one algorithm at the time). Such between-subject designs are more realistic of real world experiences. However, in our present experiment we are primarily interested in detecting differences between algorithms, some of which may be quite subtle. If users evaluated each algorithm's output separately, their experience with that algorithm would not be related to another; this is problematic as evaluation is a naturally relative activity: absolute judgments are much more difficult than relative judgments and less sensitive to small differences [HZ10]. Therefore we chose to evaluate these algorithms with a simultaneous within-subjects design in which our participants jointly evaluate two out of three algorithms side-by-side.

In internal pre-testing, the user-user and SVD algorithms often suggested very obscure movies, making it likely that they would provide recommendations that the user would be entirely unfamiliar with; while we want to measure novelty, users are limited in their ability to judge completely unfamiliar lists. There are potentially elegant solutions to this difficulty

involving learning-to-rank approaches [Liu07] and hybrid algorithms, but for our present purposes we want to test the algorithms in their pure form. Therefore, we limited each algorithm to recommending from the 2500 most-rated movies in MovieLens (about 10% of MovieLens's entire collection), making it more likely for the user to have at least heard of some of the recommended movies. This adjustment may limit effect sizes (e.g. decreasing the ability of a recommender to produce very novel recommendations), but should allow each algorithm to still demonstrate its general behaviors.

Not all algorithms could produce 10 recommendations for all users. If a user could not receive 10 recommendations from each algorithm, we exclude them from the analysis.

7.1.3 Showing Predictions

Algorithms will not necessarily generate scores on the same portions of the rating scale. For example, one algorithm may tend to predict 4.5–5 stars, while another algorithm may be more conservative and predict 3.5–4.5 stars. Since MovieLens usually shows its predicted rating with recommendations, this could have a confounding effect if the predicted rating affects the user's perception of the recommendations. To control for this, we assigned each user randomly to one of the following prediction conditions:

- Show no predictions (just the list of recommended movies).
- Show a standard, unadjusted prediction.
- Show a normalized prediction. In this condition, we predicted the first 3 movies at 5 stars, the next 4 at 4.5, and the last 3 at 4 stars.

If predicted ratings do not affect the user's perception of the recommendation lists, then there should be no difference in response between these conditions and we can average across them in the final analysis.

7.1.4 User Survey

Our survey consists of four parts. The first question, visible in fig. 7.1, asks users which list of recommendations they prefer, based on their initial impression. 5 options are available, with the extremes labeled 'Much more A than B' and 'Much more B than A'.

Following initial preference are 22 questions about various aspects of the lists, designed to measure the user's perception of the recommendation lists across five factors:

Acc Accuracy — the recommender's ability to find 'good' movies.

Sat Satisfaction — the user's overall satisfaction with the recommender and their perception of its usefulness.

Und Perceived personalization ('Understands Me') — the user's perception that the recommender understands their tastes and can effectively adapt to them.

Nov Novelty — the propensity of the recommender to suggest items with which the user is unfamiliar.

Div Diversity — the diversity of the recommended items.

For each factor, we want 4–5 questions. Factor analysis requires 3 questions for the math to work out. Extra questions give us some room for error in case one of our questions does not 'work'. A question doesn't work if it is unclear so that users do not answer it consistently, or if it evokes responses that are not sufficiently consistent with the responses to other questions in its target factor. Factor analysis will indicate how well each question measures its target factor; low-loading questions will be discarded. We want enough question so that we have enough to make the factor analysis work after discarding questions with low factor loadings.

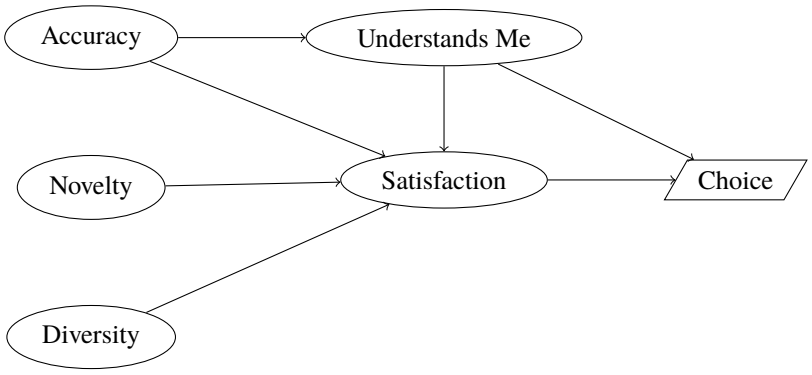


Figure 7.2: Hypothesized mediating relationships.

To develop the comparative questions, we started from the questions that Knijnenburg et al. [Kni+12] have found to work well in their previous experiments. They have published the complete list of questions, along with their target factors, from multiple experiments across multiple domains, and indicated whether each question loaded well on its factor. This gives us a bank of questions that have already been found to effectively measure user perception of recommendation in other experiments, reducing the likelihood that a question will not work. Table 7.1(a) shows the full list of questions for each factor.

We also hypothesized mediating relationships between these factors, shown in fig. 7.2. We expect the user's selection to be driven by satisfaction and perceived personalization, with personalization also having an impact on satisfaction; satisfaction, in turn, we hypothesize to be affected by the accuracy, novelty, and diversity. These hypotheses are theory-driven, coming from the psychological models of human preference and decision-making and from the relationships found in previous studies [Kni+12]. We also hoped to target a separate *serendipity* factor, a combination of novelty and accuracy that would mediate novelty's influence on satisfaction, but we had difficulty writing questions we thought would reasonably separate serendipity from novelty and the survey was already quite long with 5 latent factors.

After the main body of questions, we ask users which algorithm they would like to use by default once MovieLens gains the ability to support multiple recommender algorithms in parallel (a feature we are planning to develop in the coming months). This question is forced-choice, requiring users to pick one of the two algorithms. It also carries some consequence for users: while they will be able to switch algorithms in their user settings page without much difficulty, the algorithm they select will be providing their default recommendations in the future.

7.1.5 Objective Metrics

In addition to soliciting users' subjective perceptions of the recommendations, we computed objective measures of the algorithms' behavior with respect to accuracy, novelty, and diversity.

We estimate the accuracy of each algorithm by computing the RMSE of using it to predict each user's last 5 ratings prior to taking the survey, averaging the errors per user. To estimate novelty, we take the simple approach of computing the mean popularity rank of the items recommended to the user (fig. 7.4); this creates an 'obscurity' metric, where high values correspond to lists with more obscure items.

We compute diversity with intra-list similarity [Zie+05] using cosine between tag genome vectors [VSR12] as the itemwise similarity function and normalizing the final metric so that a list of completely similar items has a score of 1; we exclude items for which tag genome data is not available (no list required us to exclude more than 2 items); fig. 7.4 shows these values.

To convert the metrics into comparative measures, we take the log ratio of the objective metric values for the two recommendation lists presented to a user[2]. This produces a single

[2]We also experimented with computing raw differences, but generally found the log ratio to be a better

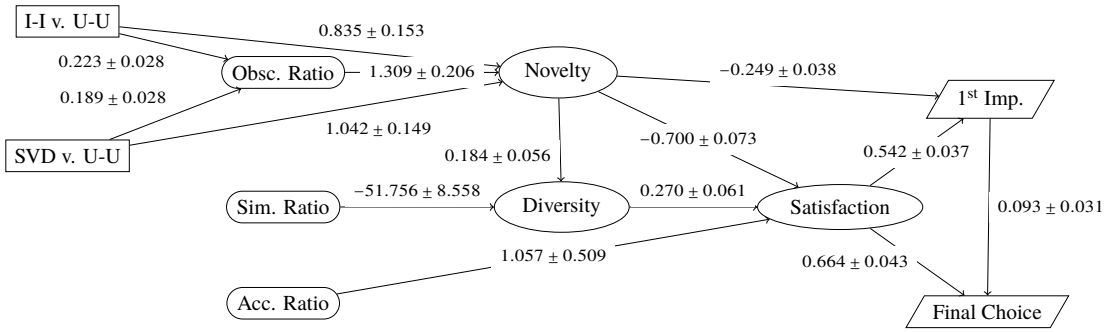


Figure 7.3: Overall SEM with bootstrapped standard errors. All displayed coefficients are significantly nonzero ($p < 0.01$). The baseline condition is I-I v. SVD; positive values & coefficients favor the right-hand algorithm (SVD or U-U).

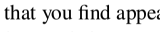
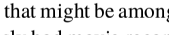
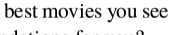
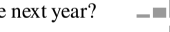
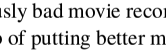
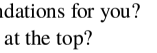
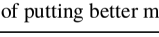
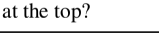
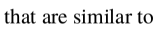
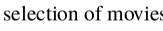
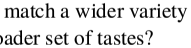
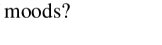
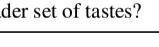
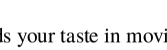
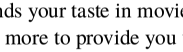
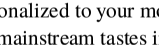
value for a pair of algorithms or recommendation lists that we can attempt to correlate with the users' subjective comparative judgements.

7.2 Results

582 users completed the study over 81 days. Table 7.2 shows how many participated in each algorithm condition, along with their final choice of algorithm. Users generally selected both item-item and SVD over user-user ($p < 0.0001$), but there was no statistically significant difference in the proportion of users choosing between item-item vs. SVD. Table 7.1(b) summarizes the responses to each of our questions by algorithm condition, and fig. 7.4 shows the objective measures of each algorithm's output.

We observed no significant effect of either the ordering of algorithms or of the prediction condition, so we exclude those from the remainder of the analysis.

predictor.

Factor / Question (*W. l.* = 'Which list', *W. r.* = 'Which recommender')	II v. SVD	II v. UU	SVD v. UU	Full CFA Coef.	Full CFA R^2	SEM Coef.
First Impression						
Accuracy					**0.61**	
W. l. has more movies that you find appealing?				0.911	0.85	
W. l. has more movies that might be among the best movies you see in the next year?				0.786	0.64	
W. l. has more obviously bad movie recommendations for you?				−0.751	0.59	
W. r. does a better job of putting better movies at the top?				0.572	0.35	
Diversity					**0.64**	
W. l. has more movies that are similar to each other?				−0.772	0.61	−0.748
W. l. has amore varied selection of movies?				0.772	0.61	0.743
W. l. has movies that match a wider variety of moods?				0.838	0.71	0.806
W. l. would suit a broader set of tastes?				0.793	0.64	0.768
Understands Me					**0.63**	
W. r. better understands your taste in movies?				0.933	0.88	
W. r. would you trust more to provide you with recommendations?				0.943	0.90	
W. r. seems more personalized to your movie ratings?				0.842	0.73	
W. r. more represents mainstream tastes instead of your own?				−0.072	0.01	
Satisfaction					**0.82**	
W. r. would better help you find movies to watch?				0.923	0.86	0.737
W. r. would you be more likely to recommend to your friends?				0.846	0.73	0.678
W. l. of recommendations do you find more valuable?				0.884	0.79	0.717
W. r. would you rather have as an app on your mobile phone?				0.921	0.86	0.736
W. r. would better help to pick satisfactory movies?				0.928	0.87	0.745
Novelty					**0.43**	
W. l. has more movies you do not expect?				0.770	0.64	0.750
W. l. has more movies that are familiar to you?				−0.784	0.66	−0.762
W. l. has more pleasantly surprising movies?				−0.454	0.24	
W. l. has more movies you would not have thought to consider?				0.704	0.54	0.707
W. l. provides fewer new suggestions?				−0.258	0.08	

(a) Questions

(b) Response distributions. Dark entries have significant bias (uncorrected Wilcox test, $p < 0.01$)

(c) CFA & SEM factor loadings.

Table 7.1: Summary of survey results. All SEM factor loadings are significant ($p < 0.01$); factor R^2 is AVE.

Condition (*A* **v.** *B*)	*N*	**Pick** *A*	**Pick** *B*	**% Pick** *B*	*p*
I-I v. U-U	201	144	57	28.4%	0.000
I-I v. SVD	198	101	97	49.0%	0.831
SVD v. U-U	183	136	47	25.7%	0.000

Table 7.2: Final algorithm selection by condition. p-values are for two-sided proportion tests, $H_0 : a/b = 0.5$.

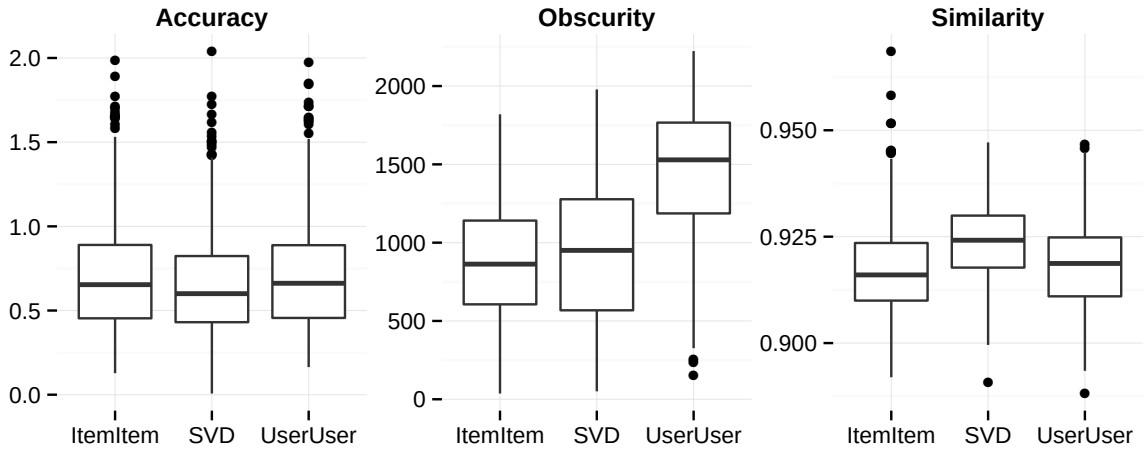


Figure 7.4: Objective recommendation list properties.

7.2.1 Response Model

To answer our more detailed research questions about the factors at play in users' choice of algorithms, we subjected the survey results to confirmatory factor analysis (CFA) and structure equation modeling (SEM). We used Lavaan [Ros12] for R [R C14] to compute the CFA and SEM, treating all question responses as ordinal variables. Each question is mapped to the factor it was designed to target. Table 7.1(c) shows the question/factor loadings from both the initial CFA and a simplified SEM derived from it. In the full CFA, there are several questions that have very low explanatory power (such as 'which recommender more represents mainstream tastes?' with $R^2 = 0.006$); in addition, the *Accuracy*, *Satisfaction*, and *Understands Me* factors are very highly correlated (correlation coefficients in

excess of 0.9), so we cannot legitimately consider them to be measuring different constructs in this experiment. We simplify the model by removing the *Accuracy* and *Understands Me* factors (we retain *Satisfaction* because it has the highest explanatory power, as measured by the Average Variance Extracted, and all 5 of its questions load strongly), and removing poorly-loading questions from *Novelty*. Also, in our theoretical model for designing the survey (fig. 7.2), satisfaction depended on both accuracy and perceived personalization. Our criteria for removing questions are based on the coefficients and R^2 values for the questions, as well as the AVE (Average Variance Extracted) for the factor. We want each factor to have high extracted variance, and each question to load strongly (high R^2) on its factor.

We then expand the simplified CFA into an SEM, which we call the *Overall SEM*, by adding structural relationships between factors, regressing them against the experimental conditions and objective metrics, and regressing the user's first impression and final selection against the experimental factors. The final SEM is built by first building a mega-SEM with many of the possible relationships, including ones that theory suggests should not exists, and then removing relationships that do not achieve statistical significance or have very small effect sizes. This gives us a structural model with significant mediating relationships between factors (e.g. the impact of diversity on choice can be explained by its impact on satisfaction). This model is hopefully consistent with our theory — and in this experiment, the theory and results line up reasonably well — but we also test for relationships that we did not expect, such as the direct effect of novelty on the user's first impression.

Figure 7.3 and table 7.1(c) show the structure and question/factor loadings in this overall model. The overall SEM has good fit ($\chi^2_{139} = 229.5$, $p < 0.001$, CFI = 0.999, TLI = 0.998, RMSEA = 0.033). The model uses standardized factor scores, so a coefficient for the effect on or of a factor measures the effect in standard deviations of the factor. We use item-item vs. SVD as the baseline condition, encoding the item-item/user-user and SVD/user-user

```
Sat =~ SatFind + SatMobile + SatRecommend + SatSat + SatValuable
Div =~ DivMoods + DivSimilar + DivTastes + DivVaried
Nov =~ NovUnexpected + NovFamiliar + NovUnthought

PopRatio ~ CondIIUU + CondSVDUU

Div ~ Nov + SimRatio
Nov ~ CondIIUU + CondSVDUU + PopRatio

Sat ~ Nov + Div + PredAccRatio
FirstImpression ~ Sat + Nov
PickedB ~ Sat + FirstImpression
```

Listing 7.5: Lavaan code for the overall SEM.

conditions with dummy condition variables. The Lavaan code to define the overall SEM is shown in listing 7.5; appendix B contains the full Lavaan output for all models.

7.2.2 RQ1: Predicting Preference

To address RQ1, we consider the impact of the factors (*Nov*, *Div*, and *Sat*) on the user's first impression of the recommendation lists and on their final choice of algorithm (see fig. 7.3). Most users who preferred one algorithm over the other at their first impression picked that algorithm in the final forced-choice question.

The only significant predictor (besides first impression) of the user's final choice of algorithm was their relative satisfaction with the two recommendation lists. Users tended to pick the algorithm with which they expressed more satisfaction.

Satisfaction in turn is influenced by the novelty (negatively) and diversity (positively) of the recommended items. Novelty also has a small positive impact on diversity, suggesting that there is an upside to novelty (as it correlating with more diverse lists, which correlates

positively with satisfaction) but a strong downside (users don't like recommendation lists full of unfamiliar items).

In addition to the its indirect effect through satisfaction, novelty had an additional negative influence on the user's first-impression preference. This means that novelty has a strong initial impact on user preference. However, after the user has made their first judgement, answered the more in-depth questions, and finally selected an algorithm, the direct impact of novelty went away and their final choice depended primarily on satisfaction. Novelty is still a significant negative influence, but it is mediated through satisfaction.

7.2.3 RQ2: Algorithm Performance

In RQ2, we want to understand how the algorithms themselves compare on relative satisfaction, diversity, novelty, and user preference as exhibited in their choice of algorithm. Table 7.2 summarizes the final choice performance of the three algorithms: as measured by users picking an algorithm for use, user-user clearly loses, and item-item and SVD are effectively tied.

Table 7.1(b) provides some insight into users' perception of the relative characteristics of the algorithms. Across most questions, item-item and SVD are indistinguishable (user responses are symmetrically distributed about the neutral response). Item-item shows slightly more diversity than SVD. The other algorithm pairings show more differences across the board, with the exception of item-item and user-user being indistinguishable on diversity.

Our overall SEM (fig. 7.3) and related factor analysis incorporate the experimental condition, but its impact is difficult to interpret due to the comparative nature of the experiment. To better understand each pair of algorithm's relative performance, we reinterpret our experiment as three between-subjects pseudo-experiments. Each of these pseudo-experiments uses one of the algorithms as a baseline and compares the other two algorithms on their per-

Baseline	Tested	% Tested > Baseline	p
ItemItem	SVD	48.99	0.0000
	UserUser	28.36	
SVD	ItemItem	51.01	0.0000
	UserUser	25.68	
UserUser	ItemItem	71.64	0.6353
	SVD	74.32	

Table 7.3: Split experiment summary. p-values are testing the null hypothesis that the user picked the tested algorithm over the baseline the same proportion of the time.

formance and behavior relative to the baseline; the experimental treatment is the choice of algorithm to compare against the baseline We will refer to this algorithm as the *tested* algorithm.

Randomization ensures that the behavior characteristics of the baseline algorithm are likely to be evenly distributed between the two sets of users encountering that algorithm, so we can (with some limitations) interpret relative measurements of one algorithm's comparison with the baseline as absolute measurements of that algorithm's behavior for the purposes of comparing with measurements of another algorithm against the same baseline.

Table 7.3 shows the layout and selected algorithm results from this interpretation. The first pair of rows describes one of the three pseudo-experiments. Examining all users assigned to one of the two conditions involving item-item CF, we use item-item as the baseline and ask how often users picked user-user or SVD over the baseline. We can apply this interpretation to all questions and factors, not just selection. This allows us to make cleaner inferences at the expense of some statistical power.

For each experiment, we re-analyzed the data using SEM and basic regressions to predict the user's relative preference and final choice. We used the factor loadings from the overall SEM and re-learned the relationship between the factors and condition, as well as

the strength of the relationships between factors themselves and their ability to predict the user's impression and choice. We also omitted the objective metrics from these SEMs in order to focus on the subjective differences between the algorithms. For brevity, we do not include a separate diagram for each experiment; the model structures are a simplification of fig. 7.3.

In addition to the condition, we also consider the number of ratings in the user's history prior to joining the experiment as a proxy for their level of experience. It is possible for algorithms to perform differently for different users, or for more experienced users to judge recommendation lists differently. We computed the median number of ratings for the users participating in the experiment and set a condition variable indicating whether a particular had 'many' or 'few' ratings relative to the median.

SVD vs. User-User

Users perceived user-user's recommendations to be more novel than SVD's (coef. 0.953, $p < 0.001$). They also reported user-user to be producing more diverse recommendation lists (coef. 0.312, $p < 0.001$). The effect on novelty was substantially stronger; combined with novelty's strong negative influence on preference, impression, and choice, users generally found SVD's recommendations more satisfactory and desirable than user-users. The effect of novelty on diversity was not present in this model; novelty only affected satisfaction directly.

As explained above, these results are from comparative judgements between the output of the tested algorithm (SVD or user-user) and the baseline algorithm (item-item). However, due to randomization, we assume that there are no important differences in item-item's output between the users comparing it against SVD and those comparing it against user-user. Therefore, we can reasonably make inferences about the relative behavior of SVD

and user-user. These results are consistent with the raw survey response data for direct comparisons between SVD and user-user (table 7.1(b)), providing further support for their validity. They are also consistent with the objective measures of obscurity and diversity (fig. 7.4)

Users selected SVD significantly more often than user-user (table 7.3), consistent with the results from users directly comparing SVD and user-user (table 7.2).

Item-Item vs. User-User

We found no significant difference in diversity between item-item and user-user CF; this is consistent with the raw results of direct comparison of these two algorithms in table 7.1(b).

User-user produced more novel recommendation lists than item-item (coef. -1.563, $p < 0.001$). This effect interacted with user experience (rating count); for high-rating users, user-user's recommendations were not as novel as they were for low-rating users. This moderating effect was small, however, and user-user was significantly more novel than item-item even for high-rating users. There was no significant difference in item-item's novelty performance between low- and high-rating users.

Item-Item vs. SVD

Item-item produced slightly more diverse recommendations than SVD (coef. -0.260, $p < 0.001$); this is consistent with the response distributions in table 7.1(b) as well as the difference in intra-list similarity (fig. 7.4). However, diversity did not have a significant influence on satisfaction in this pseudo-experiment: the only significant predictor of satisfaction was novelty.

The number of ratings the user had in their history prior to the experiment had a significant effect on the algorithm: for high-rating users, both algorithms were more novel than

user-user. Since item-item and SVD did not have significantly different perceived novelty, this effect is reflecting user-user's decreased novelty for high-rating users. Whether there is an additional increase the novelty of item-item and SVD for high-rating users, or just a decrease in user-user's novelty, is beyond this experiment's capability to measure.

7.2.4 RQ3: Objective Metrics

To address RQ3, we consider in more detail the relationships between the objective metrics and subjective factors. Figure 7.4 shows the distributions of all objective metrics we computed.

The raw distributions of novelty and diversity measurements are consistent with the user survey results. User-user produces lists with less popular (and therefore likely more novel) items than SVD or item-item. SVD tends to produce somewhat less diverse recommendation lists. All three algorithms had comparable retrospective accuracy, with SVD having a slight edge. Popularity/obscurity was the only objective metric that we found to significantly differ between conditions in the overall model.

Each objective metric was a statistically significant predictor of its corresponding subjective factor (fig. 7.3) and no other factor. This means that there is good correspondence between the subjective and objective measures of these three concepts; also, the effect of the objective measures on final choice is completely mediated by their impact on the subjective measures. All indirect effects of objective measures on final choice are significant.

This means that predictive accuracy, for example, does affect the user's final choice, but only through the increased satisfaction that it produces. Further, the impact of novelty and diversity on satisfaction means that after controlling for predictive accuracy, diversity and novelty still have significant impacts on user satisfaction.

The direct effects of condition on novelty, in addition to the effect mediated through

objective obscurity, suggest that user-user is producing lists that users perceive to be more novel beyond the sense of novelty that our objective metric can capture.

Diversity and Similarity Metrics

Tag genome similarity is not the only way that we can measure the similarity of items. We can also measure them by comparing the similarity of rating vectors or latent feature vectors, for instance. Since this experiment provides us with data on user-perceived diversity in a context where we also have access to the full ratings database and user information, we want to use the user judgements to assess the relative ability of different similarity metrics to capture notions of similarity that matter to users.

We build item similarity metrics by taking the cosine similarity over the following vector spaces:

- Tag genome
- Ratings, centered by item mean
- Latent feature vectors (extracted from the SVD recommender)

For rating-based similarity, we ignored users that did not rate both items, so the resulting similarity function is effectively the Pearson correlation between the item vectors without accounting for rating variance. All similarity functions were integrated into the same normalized intra-list similarity metric described in section 7.1.5.

All three metrics were well but not perfectly correlated; genome and rating had a correlation of 0.69, while latent feature correlation's with each was around 0.8.

Tag genome was the best single predictor of user-perceived diversity, as judged by overall model fit and strength of the coefficient's z test statistic (all regressions were significant,

Sim. Metric	Correlation			SEM Fit	
	Genome	Rating	Lat. Feat.	Div Coef.	RMSEA
Genome	1.0	0.698	0.791	-0.345	0.0335
Rating		1.0	0.821	-0.329	0.0376
Lat. Feat.			1.0	-0.276	0.0303

Table 7.4: Summary of similarity metrics for measuring diversity

$p < 0.001$). Latent features produced a slightly better overall model fit, but the direct relationship between objective similarity and user-perceived diversity was worse. Rating similarity had a worse model fit (RMSEA = 0.038) but only slightly worse direct relationship.

If both genome and rating similarities are included in a model, they both achieve significance ($p = 0.001$ and $p = 0.003$, respectively), although the overall model fit suffers significantly (RMSEA = 0.058).

This suggests that there is significant agreement between these different means of computing similarities, but the tag genome [VSR12] seems to be the best predictor of user-perceived diversity by a small margin.

Table 7.4 summarizes the relationships of diversity computations based on different similarity metrics and their impact on the measurement SEM.

7.3 Discussion

We set out to measure user perception of various interesting properties of the output of different recommender systems in a widely-studied domain. Our experiment uncovered mediation effects of novelty, diversity, satisfaction on users' choice of recommender algorithms. In this section, we highlight some of the key findings.

7.3.1 Effect of Novelty

One of the most striking things we found is that the novelty of recommended items has a significant negative impact on users' perception of a recommender's ability to satisfactorily meet their information needs. This effect was particularly strong in its impact on the user's first impression of an algorithm, and was present even though we restricted the depth of the long tail into which our algorithms could reach.

This suggests that recommender system designers should carefully watch the novelty of their system's recommendations, particularly for new users. Too many unfamiliar recommendations may give users a poor impression of a particular recommender, potentially driving them to use other systems instead. Dialing up the novelty as the user gains more experience with the system and has had more time to consider its ability to meet their needs may provide benefit, but our results cannot confirm or deny this. It is worth noting that the users in our study are not inexperienced with movie recommendation in general and MovieLens in particular, and their first impression of recommendations heavily influenced by novelty.

Our results complement the notion that that *trust-building* is an important goal of a recommender in the early stage of its relationship with its users [MRK06b], providing data on some factors that may be important in the trust-building process. They are also consistent with previous results finding that novelty is not necessarily positively correlated with user satisfaction or adoption of recommendations [CH08].

7.3.2 Diversity

We have also demonstrated that the diversity of recommendations has a positive influence on user choice of systems for general-purpose movie recommendation. Diversity is often framed as being in tension with accuracy, so that accuracy must be sacrificed in order to

obtain diverse recommendation lists [Zie+05; ZH08; Zho+10], and many diversification techniques do result in reduced accuracy by traditional objective measures. The strong correlation of perceived accuracy and satisfaction in our results provide evidence that there may not be such a trade-off when considering user perception instead of traditional accuracy metrics. This is consistent with other recent work on diversity that finds it to be a valuable component of choice [Bol+10; WGK14] and that it improves users' perception of the accuracy or quality of recommendations [Kni+12].

The influence of novelty and diversity on satisfaction even after controlling for predictive accuracy provides direct, quantitative evidence for subjective but observable characteristics of recommendation lists that affect user satisfaction and choice.

Diversity was also mildly but positively influenced by novelty.

Finally, our data suggest that the computing item similarities using the tag genome maps better to user-perceived list diversity than rating or latent feature similarity, but we do not advise relying very strongly on this result at present. Its advantage is not very strong, by what we have been able to measure thus far, and more study is needed to more directly understand the mapping of similarity functions to user perception.

7.3.3 Algorithm Performance

When it comes to comparing the particular algorithms that we tested, item-item and SVD performed very similar, with users preferring them in roughly equal measure. We do not yet have insight into whether there are identifiable circumstances in which one is preferable over the other. It may be that one works better for some users than others; it may also be that their performance is roughly equivalent, and one does not work significantly better. The difference in the diversity of SVD and item-item, however, provides evidence that the two algorithms are doing something interestingly different.

User-user is the clear loser in our tests. Its predictive accuracy was comparable to that of the other algorithms, but it had a significant propensity for novel recommendations that hurt both users' expressed satisfaction with its output and their interest in using it in the future. The lack of a significant independent effect of user-user condition on satisfaction or selection suggests that the increased novelty is the primary cause of user-user's poor subjective performance.

Finally, all three algorithms had similar predictive accuracy, but users still had strong preferences between some pairings. However, users selected item-item and SVD in almost equal numbers even though SVD had slightly higher predictive accuracy. This provides additional evidence that, at least beyond a certain point, offline metrics fail to capture much of what will impact the user's experience with a recommender system.

7.3.4 Limitations and Generalizability

This experiment does have certain limits. While the comparative setup allows us to measure nuanced differences in recommender behavior, it cannot discern between certain kinds of changes. If, for example, the difference in perceived novelty between user-user and item-item decreases for some users, we cannot tell if that is because user-user is less novel for those users, item-item is more novel, or some characteristic of those users simultaneously affects the novelty of both algorithms in opposite directions. A comparative study of this type also cannot distinguish between to recommenders being equally bad and equally good. This is an important trade-off in the experimental design, and our results should be integrated with the results from other studies (both existing and yet-to-be-done) in non-comparative settings to paint a fuller picture of the behaviors each algorithm exhibits. This is an important direction of future research.

Also, this experiment has focused on movie recommendation with experienced users,

studying three widely-used algorithms and commonly acknowledged recommendation characteristics. But we need a general understanding of algorithm behavior to meet the goals of recommender engineering. This experiment, therefore, raises as many questions as it answers. Under what circumstances, for instance, do users want diversity or eschew novelty? Does user-user generally have a high degree of novelty, or is that a property that emerges when it is applied to movies but may not be present in other domains such as conference co-attendees?

We believe the structure of our experiment and analysis provide a good starting point for generalizing and further validating our results. The structural models have allowed us to decompose users' overall choices and responses into constituent components with mediating relationships. This allows further studies in other contexts to specifically target those relationships, and should provide a framework for understanding how other contexts differ or remain the same. For example, suppose a study on another user task or with users with different characteristics find that user-user still provides exceedingly novel recommendations, but users like the recommendations it provides. Such a study would be evidence for user-user's general novelty, and that the strong negative influence of novelty is task-dependent.

Also, using the data from this and other surveys to calibrate offline metrics, as we have done with diversity metrics, will hopefully provide us with a very valuable set of tools for more general explorations in recommendation. To be sure, these metrics also need to be validated in multiple domains and tasks — a good movie diversity measurement may not be good for books, research articles, or legal briefs — but there are many readily-available means, some of which we have explored, for extending the impact of this work beyond its immediate context.

Chapter 8

Conclusion

In this thesis, we have put forward a vision of engineering recommender systems from well-understood principles and made several contributions laying a foundation for the research needed for this vision to realized. These contributions include:

- LensKit, a software package for reproducible recommender systems research, capable of recreating existing research and supporting a wide array of recommender experiments (including both offline evaluations and research projects involving deployment into production services). It enables reproducible research on a wide range of algorithms, metrics, data sets, and applications.

- Offline experiments on recommender system configuration choices and the impact of metrics on those choices. We have found new best practices for configuring classic collaborative filtering algorithms and showed that using rank accuracy instead of predictive accuracy does not have significant impact on configuration choices, but using a top-N evaluation metric does. We have also developed a systematic method for tuning item-item collaborative filtering, and reported the results of our attempts to do the same for the FunkSVD algorithm. This work provides researchers and developers with new insights into how to tune recommender algorithms, helping to decrease the search space of recommender solutions.

- An offline experiment in the errors made by different recommender algorithms, showing that having the best prediction accuracy (RMSE) does not necessarily mean that an algorithm will get the most predictions correct in a user-visible fashion, and that item-item and user-user collaborative filtering mispredict different ratings. This experiment provides additional evidence that recommender algorithms differ in ways that can possibly be exploited to improve recommender systems' ability to serve their users.

- A comparative user study of collaborative filtering algorithms, showing the following:

  - Perceived novelty and diversity affects user satisfaction with a movie recommender algorithm, and satisfaction in turn predicts the algorithm they will choose.
  - Too much novelty makes users dissatisfied with a recommender algorithm.
  - Novelty has a particularly strong negative effect on users' first impression of a recommendation list, whereas their choice after more detailed consideration of the list depends more on satisfaction.
  - Offline metrics can predict some of the user-perceived novelty, diversity, and satisfaction, but not a substantial amount.
  - User-user collaborative filtering produces substantially more novel recommendations than item-item or SVD.

  This study has also identified particular relationships between aspects of user perception of recommendations that can be tested, validated, and nuanced in further studies.

In support of our software development work, we have also developed a new approach to using dependency injection to configure object-oriented software, allowing LensKit to

support arbitrarily complex recommender configurations by composing individually simple components. This technique improves on previous dependency injection systems in two significant ways:

- It exposes component configurations as first-class objects that can be manipulated and analyzed to enable a set of rich functionality on top of the configured systems (in our application, recommender algorithms).
- Its configuration language and dependency resolution algorithm allow for components to be arbitrarily composed. Previous solutions either required component implementations to be aware of the ways in which they might be reused and provide appropriate qualifiers on their dependencies — often requiring many extra implementations for different composition scenarios — or verbose configuration.

8.1 Planned Work

In the user experiment in chapter 7, we told users that we will be supporting multiple algorithms in MovieLens. That feature will hopefully be deployed in the near future, and after it is deployed we will look at user behavior with the different recommenders. In particular, we want to know if users continue to use the recommender they said they preferred in our study over the long term, and if there are differences in user retention, activity, and rating patterns between the different algorithms.

On LensKit, there is a great deal of work to be done in building new evaluation techniques, algorithms, metrics, and improving the user experience of the software. Our next development priorities include:

- Improving ease-of-use through making better use of existing software, writing convenience APIs, and providing more tutorial documentation and examples.
- Decreasing LensKit's direct inclusion of specialized but generally-useful infrastructure, such as the data structures and evaluation script processor, by either replacing them with off-the-shelf components (we want to integrate the evaluator with the Gradle automation tool to reuse its task and project management support) or spinning them off into stand-alone projects (several of LensKit's data structures and some of the data processing code in the evaluator would be useful as general-purpose libraries and Gradle extensions).
- Improving and testing support for recommendation and evaluation with non-rating data.
- Implementing dynamic learning rates and Bayesian optimization for iterative methods.
- Finishing development of a web service to expose LensKit's capabilities to non-Java systems via an HTTP REST API.
- Integrating with GraphLab[1] to make their array of high-performance machine learning algorithms available as LensKit recommenders.

One particular problem for LensKit in its current state is the complexity of selecting, configuring and tuning a recommender algorithm. LensKit is a complex piece of software, and so far our development effort has focused on making it possible to control and configure its various options. We hope to address the ease-of-use problem by providing a library of

[1] `http://graphlab.com`

well-documented example configurations and by developing tools to make the configuration options more discoverable and better-documented. A GUI tool for viewing and creating configurations that allows the user to see the available options for different configuration points would be very helpful. Further work on automatic parameter tuning would also help alleviate some of the configuration burden.

8.2 Further Ideas

The vision of recommender engineering requires a great deal of further research. We need to explore what makes recommendation successful in a wide range of domains and tasks. These studies need to be designed to understand *why* a particular recommendation approach performs well or poorly, not just that it does. Understanding mediating reasons for recommender performance — behavior of the algorithm, properties of the domain, characteristics of the user or task — will greatly aid in generalizing the results and developing a systematic science of recommendation. This can be done in both online and offline settings. Carefully-designed user studies, factor analysis, structural equation modeling are powerful tools for understanding the 'why' of user perception of the recommender. Offline experiments, though, can also be very useful if they are designed to elucidate what it is that different algorithms do differently. Comparing solely on accuracy does not paint a broad picture of algorithm performance and suitability.

One promising technique for better understanding recommender behavior in offline settings is temporal evaluation [LHC09]. We hope to add temporal evaluation support to LensKit in coming years and investigate more deeply the relative behaviors of different algorithms at different points in the lives of users, items, and systems. There is also much more work to be done to develop offline experiment methodologies and metrics to answer

interesting questions about algorithm behavior.

We would also like to see the problem of systematic parameter tuning explored more thoroughly, and eventually automated. Efficient, automatic mechanisms for optimizing recommenders and tuning their hyperparameters will reduce the search space that a recommender developer must consider. One key question that remains is the use of subsampling: can some parameters be tuned by using a subset of the data? If so, what ones?

8.3 The Road Ahead

The work presented in this thesis is only the starting point of the research necessary to put recommender engineering and development on a solid, well-understood footing. We hope, through further development on and research with LensKit along with additional experiments in running systems, to continue to contribute to this work. It will also require substantial work from researchers with access to other user bases in other applications and domains.

Even if it ultimately turns out that recommendation is too noisy of a problem, so that we can never arrive at a definitive understanding of how to build optimal solutions to arbitrary recommendation tasks, it is our opinion that the goal is still worthwhile. The research needed to take us in that direction — indeed, to determine that the problem is, in the long run, insoluble — will greatly improve our understanding of how people interact with recommender systems (and related systems, such as information retrieval and information filtering tools), and factors that influence the success or failure of particular recommender applications. We look forward with great anticipation to the discoveries yet to be made in pursuit of a systematic science of recommendation.

Bibliography

[Ama11] Xavier Amatriain. *Recommender Systems: We're doing it (all) wrong*. Technocalifornia. Apr. 7, 2011. URL: http://technocalifornia.blogspot.com/2011/04/recommender-systems-were-doing-it-all.html (visited on 05/17/2011).

[Ama12] Xavier Amatriain. *Netflix Recommendations: Beyond the 5 stars (Part 1)*. The Netflix Tech Blog. Apr. 6, 2012. URL: http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html (visited on 05/21/2014).

[AMT05] Paolo Avesani, Paolo Massa, and Roberto Tiella. "A trust-enhanced recommender system application: Moleskiing". In: *ACM SAC '05*. ACM, 2005, pp. 1589–1593. ISBN: 1-58113-964-0. DOI: 10.1145/1066677.1067036.

[Apa11] Apache Software Foundation. *Lucene*. Version 3.5.0. Nov. 2011.

[AZ12] Gediminas Adomavicius and Jingjing Zhang. "Stability of Recommendation Algorithms". In: *ACM Trans. Inf. Syst.* 30.4 (Nov. 2012), 23:1–23:31. ISSN: 1046-8188. DOI: 10.1145/2382438.2382442.

[BDO95] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. "Using Linear Algebra for Intelligent Information Retrieval". In: *SIAM Review* 37.4 (Dec. 1995), pp. 573–595. ISSN: 00361445.

[Bel12] Alejandro Bellogin. "Performance prediction and evaluation in Recommender Systems: an Information Retrieval perspective". Doctoral Thesis. Madrid, Spain: Universidad Autónoma de Madrid, Nov. 2012.

[BHK98] John S Breese, David Heckerman, and Carl Kadie. "Empirical Analysis of Predictive Algorithms for Collaborative Filtering". In: *Proc. UAI '98*. 1998, pp. 43–52.

[BL07] James Bennett and Stan Lanning. "The Netflix Prize". In: *Proc. of KDD Work on Large-Scale Rec. Sys.* 2007.

[Bla03] Norman W. H. Blaikie. *Analyzing quantitative data: from description to explanation*. SAGE, Mar. 6, 2003. 376 pp. ISBN: 9780761967583.

[Blo08] Joshua Bloch. *Effective Java*. 2nd Edition. Upper Saddle River, NJ: Addison-Wesley, 2008. 346 pp. ISBN: 9780321356680 0321356683 0201310058 9780201310054.

[BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation". In: *J. Mach. Learn. Res.* 3 (Mar. 1, 2003), pp. 993–1022.

[Bol+10] Dirk Bollen et al. "Understanding choice overload in recommender systems". In: *Proceedings of the fourth ACM conference on Recommender systems*. RecSys '10. ACM ID: 1864724. New York, NY, USA: ACM, 2010, pp. 63–70. ISBN: 978-1-60558-906-0. DOI: 10.1145/1864708.1864724.

[BS97] Marko Balabanović and Yoav Shoham. "Fab: content-based, collaborative recommendation". In: *Commun. ACM* 40.3 (1997), pp. 66–72. DOI: 10.1145/245108.245124.

[Bur02] Robin Burke. "Hybrid Recommender Systems: Survey and Experiments". In: *User Modeling and User-Adapted Interaction* 12.4 (Nov. 2002), pp. 331–370. ISSN: 0924-1868. DOI: 10.1023/A:1021240730564.

[Bur10] Robin Burke. "Evaluating the dynamic properties of recommendation algorithms". In: *ACM RecSys '10*. ACM, 2010, pp. 225–228. ISBN: 978-1-60558-906-0. DOI: 10.1145/1864708.1864753.

[Can02] John Canny. "Collaborative filtering with privacy via factor analysis". In: *ACM SIGIR '02*. ACM, 2002, pp. 238–245. ISBN: 1-58113-561-0. DOI: 10.1145/564376.564419.

[CG98] Jaime Carbonell and Jade Goldstein. "The Use of MMR, Diversity-based Reranking for Reordering Documents and Producing Summaries". In: *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '98. New York, NY, USA: ACM, 1998, pp. 335–336. ISBN: 1-58113-015-5. DOI: 10.1145/290941.291025.

[CG99] Y. H Chien and E. I George. "A bayesian model for collaborative filtering". In: *7th International Workshop on Artificial Intelligence and Statistics*. 1999.

[CH08] Òscar Celma and Perfecto Herrera. "A New Approach to Evaluating Novel Recommendations". In: *Proceedings of the 2008 ACM Conference on Recommender Systems*. RecSys '08. New York, NY, USA: ACM, 2008, pp. 179–186. ISBN: 978-1-60558-093-7. DOI: 10.1145/1454008.1454038.

[Che+12] Tianqi Chen et al. "SVDFeature: A Toolkit for Feature-based Collaborative Filtering". In: *J. Mach. Learn. Res.* 13.1 (Dec. 2012), pp. 3619–3622. ISSN: 1532-4435.

[CI05] Shigeru Chiba and Rei Ishikawa. “Aspect-Oriented Programming Beyond Dependency Injection”. In: *ECOOP 2005 - Object-Oriented Programming*. Vol. 3586. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 121–143. ISBN: 978-3-540-27992-1, 978-3-540-31725-8.

[Cla+08] Charles L.A. Clarke et al. “Novelty and Diversity in Information Retrieval Evaluation”. In: *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’08. New York, NY, USA: ACM, 2008, pp. 659–666. ISBN: 978-1-60558-164-4. DOI: 10.1145/1390334.1390446.

[Dee+90] Scott Deerwester et al. “Indexing by Latent Semantic Analysis”. In: *Journal of the American Society for Information Science* 41.6 (1990), pp. 391–407. DOI: 10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9.

[DK04] Mukund Deshpande and George Karypis. “Item-based top-N recommendation algorithms”. In: *ACM Transactions on Information Systems* 22.1 (2004), pp. 143–177. DOI: 10.1145/963770.963776.

[Don+09] David L. Donoho et al. “Reproducible Research in Computational Harmonic Analysis”. In: *Computing in Science & Engineering* 11.1 (Jan. 1, 2009), pp. 8–18. ISSN: 1521-9615. DOI: 10.1109/MCSE.2009.15.

[Dow+14] M. Dowle et al. *data.table: Extension of data.frame*. Version 1.9.2. Feb. 27, 2014.

[Eks+10] Michael Ekstrand et al. “Automatically building research reading lists”. In: *RecSys ’10*. ACM, 2010, pp. 159–166. DOI: 10.1145/1864708.1864740.

[Eks+11] Michael Ekstrand et al. “Rethinking the recommender research ecosystem: reproducibility, openness, and LensKit”. In: *RecSys ’11*. ACM, 2011, pp. 133–140. ISBN: 978-1-4503-0683-6. DOI: 10.1145/2043932.2043958.

[Eks+14] Michael D. Ekstrand et al. “User Perception of Differences in Recommender Algorithms”. In: *Proceedings of the 8th ACM Conference on Recommender Systems (RecSys ’14)*. ACM, Oct. 2014.

[Eks14] Michael D. Ekstrand. “Building Open-Source Tools for Reproducible Research and Education”. In: *Proceedings of the Workshop on Sharing, Re-use and Circulation of Resources in Cooperative Scientific Work at ACM CSCW ’14*. Feb. 2014.

[ER12] Michael Ekstrand and John Riedl. “When recommenders fail: predicting recommender failure for algorithm selection and combination”. In: *Proceedings of the sixth ACM conference on Recommender systems*. RecSys ’12. New York, NY, USA: ACM, 2012, pp. 233–236. ISBN: 978-1-4503-1270-7. DOI: 10.1145/2365952.2366002.

[ERK10] Michael Ekstrand, John Riedl, and Joseph A. Konstan. “Collaborative Filtering Recommender Systems”. In: *Foundations and Trends® in Human-Computer Interaction* 4.2 (2010), pp. 81–173. ISSN: 1551-3955. DOI: 10.1561/1100000009.

[Fow04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. Jan. 23, 2004. URL: http://martinfowler.com/articles/injection.html (visited on 08/01/2011).

[Fun06] Simon Funk. *Netflix Update: Try This at Home*. The Evolution of Cybernetics. Dec. 11, 2006. URL: http://sifter.org/~simon/journal/20061211.html (visited on 04/08/2010).

[Gam+95a] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995. 395 pp.

[Gam+95b] Erich Gamma et al. “Strategy”. In: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, pp. 315–323. ISBN: 0201633612.

[Gan+11] Zeno Gantner et al. “MyMediaLite: A Free Recommender System Library”. In: *Proceedings of the Fifth ACM Conference on Recommender Systems*. RecSys ’11. New York, NY, USA: ACM, 2011, pp. 305–308. ISBN: 978-1-4503-0683-6. DOI: 10.1145/2043932.2043989.

[GDJ10] 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*. RecSys ’10. New York, NY, USA: ACM, 2010, pp. 257–260. ISBN: 978-1-60558-906-0. DOI: 10.1145/1864708.1864761.

[GL04] Robert Gentleman and Duncan Temple Lang. “Statistical Analyses and Reproducible Research”. In: *Bioconductor Project Working Papers* (May 29, 2004).

[Gof64] William Goffman. “A searching procedure for information retrieval”. In: *Information Storage and Retrieval* 2.2 (July 1964), pp. 73–78. ISSN: 0020-0271. DOI: 10.1016/0020-0271(64)90006-3.

[Gol+01] Ken Goldberg et al. “Eigentaste: A Constant Time Collaborative Filtering Algorithm”. In: *Information Retrieval* 4.2 (July 1, 2001), pp. 133–151. DOI: 10.1023/A:1011419012209.

[GS09] Asela Gunawardana and Guy Shani. “A Survey of Accuracy Evaluation Metrics of Recommendation Tasks”. In: *J. Mach. Learn. Res.* 10 (2009), pp. 2935–2962.

[Hal+09] Mark Hall et al. “The WEKA Data Mining Software: An Update”. In: *SIGKDD Explorations* 11.1 (2009).

[Her+04] Jonathan Herlocker et al. “Evaluating collaborative filtering recommender systems”. In: *ACM Trans. Inf. Syst.* 22.1 (2004), pp. 5–53. DOI: 10.1145/963770.963772.

[Her+99] Jonathan Herlocker et al. “An algorithmic framework for performing collaborative filtering”. In: *Proceedings of the 22nd annual international ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’99. ACM, 1999, pp. 230–237. DOI: 10.1145/312624.312682.

[Hil+95] William Hill et al. “Recommending and evaluating choices in a virtual community of use”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. Denver, Colorado, United States: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 194–201. ISBN: 0-201-84705-1. DOI: 10.1145/223904.223929.

[HKR02] Jonathan Herlocker, Joseph A. Konstan, and John Riedl. “An Empirical Analysis of Design Choices in Neighborhood-Based Collaborative Filtering Algorithms”. In: *Inf. Retr.* 5.4 (2002), pp. 287–310. DOI: 10.1023/A:1020443909834.

[Hof04] Thomas Hofmann. “Latent semantic models for collaborative filtering”. In: *ACM Transactions on Information Systems* 22.1 (2004), pp. 89–115. DOI: 10.1145/963770.963774.

[Hor99] Eric Horvitz. “Principles of mixed-initiative user interfaces”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI’99*. Pittsburgh, Pennsylvania, United States: ACM, 1999, pp. 159–166. ISBN: 0-201-48559-1. DOI: 10.1145/302979.303030.

[HZ10] Christopher K. Hsee and Jiao Zhang. “General Evaluability Theory”. In: *Perspectives on Psychological Science* 5.4 (July 1, 2010), pp. 343–355. ISSN: 1745-6916, 1745-6924. DOI: 10.1177/1745691610374586.

[JK02] Kalervo Järvelin and Jaana Kekäläinen. “Cumulated gain-based evaluation of IR techniques”. In: *ACM Trans. Inf. Syst. (TOIS)* 20.4 (Oct. 2002), pp. 422–446. ISSN: 1046-8188. DOI: 10.1145/582415.582418.

[JL09] Rob Johnson and Bob Lee. *JSR 330: Dependency Injection for Java*. 330. Java Community Process, Oct. 13, 2009.

[JZM04] Xin Jin, Yanzan Zhou, and Bamshad Mobasher. “Web usage mining based on probabilistic latent semantic analysis”. In: *ACM KDD ’04*. ACM, 2004, pp. 197–205. ISBN: 1-58113-888-1. DOI: 10.1145/1014052.1014076.

[KA13] Joseph A. Konstan and Gediminas Adomavicius. “Toward Identification and Adoption of Best Practices in Algorithmic Recommender Systems Research”. In: *Proceedings of the International Workshop on Reproducibility and Replication in Recommender Systems Evaluation*. RepSys ’13. New York, NY, USA: ACM, 2013, pp. 23–28. ISBN: 978-1-4503-2465-6. DOI: 10.1145/2532508.2532513.

[Kli98] Rex B. Kline. *Principles and practice of structural equation modeling*. New York: Guilford Press, 1998. 354 pp. ISBN: 1572303360.

[Klu+12] Daniel Kluver et al. “How many bits per rating?” In: *Proceedings of the sixth ACM conference on Recommender systems*. RecSys ’12. New York, NY, USA: ACM, 2012, pp. 99–106. ISBN: 978-1-4503-1270-7. DOI: 10.1145/2365952.2365974.

[Kni+12] Bart Knijnenburg et al. “Explaining the user experience of recommender systems”. In: *User Modeling and User-Adapted Interaction* 22.4 (Oct. 1, 2012), pp. 441–504. ISSN: 0924-1868, 1573-1391. DOI: 10.1007/s11257-011-9118-4.

[Koh12] Nate Kohari. *Ninject*. 2012.

[Kon+14] Joseph A. Konstan et al. “Teaching Recommender Systems at Large Scale: Evaluation and Lessons Learned from a Hybrid MOOC”. In: *Proceedings of the First ACM Conference on Learning @ Scale Conference*. L@S ’14. New York, NY, USA: ACM, 2014, pp. 61–70. ISBN: 978-1-4503-2669-8. DOI: 10.1145/2556325.2566244.

[KS11] Yehuda Koren and Joe Sill. “OrdRec: an ordinal model for predicting personalized item rating distributions”. In: *Proceedings of the fifth ACM conference on Recommender systems*. RecSys ’11. New York, NY, USA: ACM, 2011, pp. 117–124. ISBN: 978-1-4503-0683-6. DOI: 10.1145/2043932.2043956.

[Lai+07] Christine Laine et al. “Reproducible Research: Moving toward Research the Public Can Really Trust”. In: *Annals of Internal Medicine* 146.6 (Mar. 20, 2007), pp. 450–453. ISSN: 0003-4819. DOI: 10.7326/0003-4819-146-6-200703200-00154.

[LHC09] Neal Lathia, Stephen Hailes, and Licia Capra. “Evaluating Collaborative Filtering Over Time”. In: *SIGIR ’09 Workshop on the Future of IR Evaluation*. July 2009.

[Liu07] Tie-Yan Liu. “Learning to Rank for Information Retrieval”. In: *Foundations and Trends® in Information Retrieval* 3.3 (2007), pp. 225–331. ISSN: 1554-0669, 1554-0677. DOI: 10.1561/1500000016.

[LM05] Daniel Lemire and Anna Maclachlan. “Slope One Predictors for Online Rating-Based Collaborative Filtering”. In: *Proceedings of SIAM Data Mining (SDM’05)*. 2005.

[Lou03] R. Lougee-Heimer. “The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community”. In: *IBM Journal of Research and Development* 47.1 (Jan. 2003), pp. 57–66. ISSN: 0018-8646. DOI: 10.1147/rd.471.0057.

[LR04] 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. New York, NY, USA: ACM, 2004, pp. 393–402. ISBN: 1-58113-844-X. DOI: 10.1145/988672.988726.

[LSY03] G. Linden, B. Smith, and J. York. “Amazon.com recommendations: item-to-item collaborative filtering”. In: *IEEE Internet Computing* 7.1 (2003), pp. 76–80. ISSN: 1089-7801.

[Mag+95] Jeff Magee et al. “Specifying distributed software architectures”. In: *Software Engineering — ESEC ’95*. Ed. by Wilhelm Schäfer and Pere Botella. Lecture Notes in Computer Science 989. Springer Berlin Heidelberg, Jan. 1, 1995, pp. 137–153. ISBN: 978-3-540-60406-8, 978-3-540-45552-3.

[Mar96] Robert C. Martin. “The Dependency Inversion Principle”. In: *C++ Report* 8.6 (May 1996).

[Mcn+02] Sean Mcnee et al. “On the recommending of citations for research papers”. In: *CSCW ’02: Proceedings of the 2002 ACM conference on Computer supported cooperative work*. ACM, 2002, pp. 116–125. DOI: 10.1145/587078.587096.

[MKK06] Sean McNee, Nishikant Kapoor, and Joseph A. Konstan. “Don’t Look Stupid: Avoiding Pitfalls When Recommending Research Papers”. In: *Proceedings of the 2006 20th anniversary conference on Computer Supported Cooperative Work*. CSCW ’06. Banff, Alberta, Canada: ACM, 2006, p. 171. DOI: 10.1145/1180875.1180903.

[MR00] Raymond J. Mooney and Loriene Roy. “Content-based book recommending using learning for text categorization”. In: *Proceedings of the fifth ACM conference on Digital libraries*. San Antonio, Texas, United States: ACM, 2000, pp. 195–204. ISBN: 1-58113-231-X. DOI: 10.1145/336597.336662.

[MRK06a] Sean 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*. Montréal, Québec, Canada: ACM, 2006, pp. 1097–1101. ISBN: 1-59593-298-4. DOI: 10.1145/1125451.1125659.

[MRK06b] Sean McNee, John Riedl, and Joseph A. Konstan. “Making recommendations better: an analytic model for human-recommender interaction”. In: *CHI ’06 Extended Abstracts*. ACM, 2006, pp. 1103–1108. ISBN: 1-59593-298-4. DOI: 10.1145/1125451.1125660.

[MZ09] Benjamin M. Marlin and Richard S. Zemel. “Collaborative prediction and ranking with non-random missing data”. In: *ACM RecSys ’09*. ACM, 2009, pp. 5–12. ISBN: 978-1-60558-435-5. DOI: 10.1145/1639714.1639717.

[Ngu+13] Tien T. Nguyen et al. “Rating Support Interfaces to Improve User Experience and Recommender Accuracy”. In: *Proceedings of the 7th ACM Conference on Recommender Systems*. RecSys ’13. New York, NY, USA: ACM, 2013, pp. 149–156. ISBN: 978-1-4503-2409-0. DOI: 10.1145/2507157.2507188.

[Pas+10] Erick B. Passos et al. “Smart composition of game objects using dependency injection”. In: *Comput. Entertain.* 7.4 (Jan. 2010), 53:1–53:15. ISSN: 1544-3574. DOI: 10.1145/1658866.1658872.

[Pat07] Arkadiusz Paterek. “Improving regularized singular value decomposition for collaborative filtering”. In: *KDD Cup and Workshop 2007*. Aug. 2007.

[PDZ06] Roger D. Peng, Francesca Dominici, and Scott L. Zeger. “Reproducible Epidemiologic Research”. In: *American Journal of Epidemiology* 163.9 (May 1, 2006), pp. 783–789. ISSN: 0002-9262, 1476-6256. DOI: 10.1093/aje/kwj093.

[Pen+00] David Pennock et al. “Collaborative Filtering by Personality Diagnosis: A Hybrid Memory- and Model-Based Approach”. In: *In Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI-2000)*. Morgan Kaufmann, 2000, pp. 473–480.

[Pen11] Roger D. Peng. “Reproducible Research in Computational Science”. In: *Science (New York, N.y.)* 334.6060 (Dec. 2, 2011), pp. 1226–1227. ISSN: 0036-8075. DOI: 10.1126/science.1213847.

[R C14] R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2014.

[Res+94] Paul Resnick et al. “GroupLens: an open architecture for collaborative filtering of netnews”. In: *ACM CSCW ’94*. ACM, 1994, pp. 175–186. ISBN: 0-89791-689-1. DOI: 10.1145/192844.192905.

[Ric+10] Francesco Ricci et al., eds. *Recommender Systems Handbook*. Springer, 2010.

[Ric79] Elaine Rich. “User modeling via stereotypes”. In: *Cognitive Science* 3.4 (Oct. 1979), pp. 329–354. ISSN: 0364-0213.

[Rij79] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 1979. ISBN: 0408709294.

[RJ07] Ekaterina Razina and David Janzen. “Effects of Dependency Injection on Maintainability”. In: *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA* (Nov. 19, 2007), pp. 7–12.

[Ros12] Yves Rosseel. “lavaan: An R Package for Structural Equation Modeling”. In: *Journal of Statistical Software* 48.2 (2012), pp. 1–36. ISSN: 1548-7660.

[Sal92] Gerard Salton. “The state of retrieval system evaluation”. In: *Information Processing & Management* 28.4 (July 1992), pp. 441–449. ISSN: 0306-4573. DOI: 10.1016/0306-4573(92)90002-H.

[SAM13] Mohamed Sarwat, James Avery, and Mohamed F. Mokbel. “RecDB in Action: Recommendation Made Easy in Relational Databases”. In: *Proc. VLDB Endow.* 6.12 (Aug. 2013), pp. 1242–1245. ISSN: 2150-8097. DOI: 10.14778/2536274.2536286.

[Sar+01] Badrul Sarwar et al. “Item-based collaborative filtering recommendation algorithms”. In: *ACM WWW ’01*. ACM, 2001, pp. 285–295. ISBN: 1-58113-348-0. DOI: 10.1145/371920.372071.

[Sar+02] B. M. Sarwar et al. “Incremental SVD-Based Algorithms for Highly Scaleable Recommender Systems”. In: *ICCIT 2002*. ICCIT. 2002.

[SB10] Hanhuai Shan and Arindam Banerjee. “Generalized Probabilistic Matrix Factorizations for Collaborative Filtering”. In: *Data Mining, IEEE International Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 1025–1030. DOI: http://doi.ieeecomputersociety.org/10.1109/ICDM.2010.116.

[SBM12] Sebastian Schelter, Christoph Boden, and Volker Markl. “Scalable Similarity-based Neighborhood Methods with MapReduce”. In: *Proceedings of the Sixth ACM Conference on Recommender Systems*. RecSys ’12. New York, NY, USA: ACM, 2012, pp. 163–170. ISBN: 978-1-4503-1270-7. DOI: 10.1145/2365952.2365984.

[Sch+13] Sebastian Schelter et al. “Distributed Matrix Factorization with Mapreduce Using a Series of Broadcast-joins”. In: *Proceedings of the 7th ACM Conference on Recommender Systems*. RecSys ’13. New York, NY, USA: ACM, 2013, pp. 281–284. ISBN: 978-1-4503-2409-0. DOI: 10.1145/2507157.2507195.

[SHB05] Guy Shani, David Heckerman, and Ronen I. Brafman. “An MDP-based recommender system”. In: *Journal of Machine Learning Research* 6 (2005), pp. 1265–1295. ISSN: 1533-7928.

[Sil+09] Joseph Sill et al. “Feature-Weighted Linear Stacking”. In: *arXiv:0911.0460* (Nov. 3, 2009).

[Sin+13] Tobias Sing et al. *ROCR: Visualizing the performance of scoring classifiers*. Version 1.0-5. May 16, 2013.

[SKR01] J. Ben Schafer, Joseph A. Konstan, and John Riedl. “E-Commerce Recommendation Applications”. In: *Data Mining and Knowledge Discovery* 5.1 (Jan. 1, 2001), pp. 115–153. DOI: 10.1023/A:1009804230409.

[SM08] Ruslan Salakhutdinov and Andriy Mnih. “Probabilistic Matrix Factorization”. In: *Advances in Neural Information Processing Systems*. Vol. 20. Cambridge, MA: MIT Press, 2008, pp. 1257–1264.

[SM95] Upendra Shardanand and Pattie Maes. “Social information filtering: algorithms for automating “word of mouth””. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. Denver, Colorado, United States: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 210–217. ISBN: 0-201-84705-1. DOI: 10.1145/223904.223931.

[SMH07] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. “Restricted Boltzmann machines for collaborative filtering”. In: *ACM ICML ’07*. ACM, 2007, pp. 791–798. ISBN: 978-1-59593-793-3. DOI: 10.1145/1273496.1273596.

[SVR09] Shilad Sen, Jesse Vig, and John Riedl. “Tagommenders: connecting users to items through tags”. In: *Proceedings of the 18th international conference on World wide web*. Madrid, Spain: ACM, 2009, pp. 671–680. ISBN: 978-1-60558-487-4. DOI: 10.1145/1526709.1526800.

[Swe63] John A. Swets. “Information Retrieval Systems”. In: *Science* 141.3577 (July 19, 1963), pp. 245–250. ISSN: 00368075.

[Tan11] O. Tange. “GNU Parallel — The Command-Line Power Tool”. In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47.

[Tor+04] Roberto Torres et al. “Enhancing digital libraries with TechLens+”. In: *Proceedings of the 4th ACM/IEEE-CS joint conference on Digital libraries*. Tuscon, AZ, USA: ACM, 2004, pp. 228–236. ISBN: 1-58113-832-6. DOI: 10.1145/996350.996402.

[VC11] 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*. RecSys ’11. New York, NY, USA: ACM, 2011, pp. 109–116. ISBN: 978-1-4503-0683-6. DOI: 10.1145/2043932.2043955.

[VKV09] P. Vandewalle, J. Kovacevic, and M. Vetterli. “Reproducible research in signal processing”. In: *IEEE Signal Processing Magazine* 26.3 (May 2009), pp. 37–47. ISSN: 1053-5888. DOI: 10.1109/MSP.2009.932122.

[VSR12] Jesse Vig, Shilad Sen, and John Riedl. “The Tag Genome: Encoding Community Knowledge to Support Novel Interaction”. In: *ACM Trans. Interact. Intell. Syst.* 2.3 (Sept. 2012), 13:1–13:44. ISSN: 2160-6455. DOI: 10.1145/2362394.2362395.

[WC14] Hadley Wickham and Winston Chang. *ggplot2: An implementation of the Grammar of Graphics*. Version 1.0.0. May 21, 2014.

[WF14] Hadley Wickham and Romain Francois. *dplyr: dplyr: a grammar of data manipulation*. Version 0.2. May 21, 2014.

[WGK14] Martijn C. Willemsen, Mark P. Graus, and Bart P. Knijnenburg. “Understanding the Role of Latent Feature Diversification on Choice Difficulty and Satisfaction”. In: *Under review.* (2014).

[Wic14a] Hadley Wickham. *plyr: Tools for splitting, applying and combining data*. Version 1.8.1. Feb. 26, 2014.

[Wic14b] Hadley Wickham. *reshape2: Flexibly reshape data: a reboot of the reshape package*. Version 1.4. Apr. 23, 2014.

[YL99] Yiming Yang and Xin Liu. “A re-examination of text categorization methods”. In: *ACM SIGIR ’99*. ACM, 1999, pp. 42–49. ISBN: 1-58113-096-1. DOI: 10.1145/312624.312647.

[YTM08] Hong Yul Yang, E. Tempero, and H. Melton. “An Empirical Study into Use of Dependency Injection in Java”. In: *19th Australian Conference on Software Engineering, 2008. ASWEC 2008*. 19th Australian Conference on Software Engineering, 2008. ASWEC 2008. IEEE, Mar. 26, 2008, pp. 239–247. ISBN: 978-0-7695-3100-7. DOI: 10.1109/ASWEC.2008.4483212.

[ZH08] Mi Zhang and Neil Hurley. “Avoiding Monotony: Improving the Diversity of Recommendation Lists”. In: *Proceedings of the 2008 ACM Conference on Recommender Systems*. RecSys ’08. New York, NY, USA: ACM, 2008, pp. 123–130. ISBN: 978-1-60558-093-7. DOI: 10.1145/1454008.1454030.

[Zho+10] Tao Zhou et al. “Solving the apparent diversity-accuracy dilemma of recommender systems”. In: *Proceedings of the National Academy of Sciences* 107.10 (Mar. 9, 2010), pp. 4511–4515. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.1000488107.

[Zie+05] Cai-Nicolas Ziegler et al. “Improving recommendation lists through topic diversification”. In: *Proceedings of the 14th international conference on World Wide Web*. Chiba, Japan: ACM, 2005, pp. 22–32. ISBN: 1-59593-046-9. DOI: 10.1145/1060745.1060754.

[ZZ06] Philip Zigoris and Yi Zhang. “Bayesian adaptive user profiling with explicit & implicit feedback”. In: *ACM CIKM ’06*. ACM, 2006, pp. 397–404. ISBN: 1-59593-433-2.

Appendix A

LensKit Manual Pages

These are the manual pages for the LensKit command line tools.

A.1 lenskit

Name

lenskit - a command-line tool for LensKit

Synopsis

lenskit [OPTIONS] *subcommand* [arguments]

Description

The LensKit command line tool provides several capabilities for examining, evaluating, and using LensKit recommender algorithms. It primarily operates on LensKit algorithm configurations and eval scripts written in the corresponding Groovy DSLs.

The various specific tools are exposed via subcommands, much like **git**(1) and similar tools. The subcommands are listed below (see Subcommands), and each is described in more detail in its own manual page.

Options

--help Print usage instructions.

--log-file *FILE* Write logging output to *FILE*.

-d, --debug Increase verbosity, printing debug messages to the console. By default, only messages at INFO and higher levels are logged. The log file, if specified, always receives debug-level output.

--debug-grapht Output INFO (or DEBUG, if **--debug** is also used) logging messages from Grapht. Grapht is pretty noisy, so by default its output is filtered to warnigns and errors. If you need to debug a problem that is occurring in Grapht, use this option.

Subcommands

Each command is documented in its own man page, *lenskit-command*(1).

version Print the LensKit version.

train-model Train a recommender model and save it to disk.

predict Predict user ratings for items, using a configuration or a trained model.

recommend Recommend items for users, using a configuration or a trained model.

graph Output a GraphViz diagram of a recommender configuration (either from configuration files or a trained model).

eval Run a LensKit evaluation script.

pack-ratings Pack rating data into a binary file for more efficient access.

Environment and System Properties

The LensKit CLI (or its launcher script) recognize the following environment variables:

JAVA_OPTS Additional flags to pass to the JVM (such as `-Xmx4g` to set the memory limit).

JAVA_HOME Where to find the Java Runtime Environment.

Also, the following Java system properties can be set for useful effects:

logback.configurationFile The location of a Logback configuration file. This overrides all built-in or command line logging configuration.

See Also

- Man pages for subcommands: **lenskit-version**(1), **lenskit-train-model**(1), **lenskit-predict**(1), **lenskit-recommend**(1), **lenskit-graph**(1), **lenskit-eval**(1), **lenskit-pack-ratings**(1)
- The LensKit home page
- The LensKit manual

A.2 lenskit-version

Name

lenskit version - print LensKit version info.

Synopsis

lenskit [GLOBAL OPTIONS] **version**

Description

The **version** command prints the current version of LensKit and exits.

This subcommand takes no arguments.

See Also

lenskit(1)

A.3 lenskit-train-model

Name

lenskit train-model - train a LensKit model and serialize it to disk.

Synopsis

lenskit [GLOBAL OPTIONS] **train-model** [OPTIONS] *CONFIG*...

Description

The `train-model` command loads a LensKit algorithm configuration, instantiates its shareable components, and writes the resulting recommender engine to a file. This file can then be loaded into an application or one of the other LensKit commands to provide recommendations and predictions.

Options

CONFIG A LensKit algorithm configuration file, written in the LensKit algorithm DSL for Groovy. If multiple configuration files are specified, they are used together, with configuration in later files taking precedence over earlier files.

--help Show usage help.

-o *FILE*, --output-file *FILE* Write the resulting recommender model to *FILE*. If this option is not specified, the model will be written to `model.bin` in the current directory. If *FILE* ends in `.gz`, the file will be gzip-compressed. Compressed model files can be transparently read by LensKit, so this is usually a good idea.

Input Data Options

This command can read data in several different ways. To give the model building process some data to work with, one of the following mutually-exclusive options must be present:

--ratings-file *FILE* Read ratings from the delimited text file *FILE*.

--csv-file *FILE* Read ratings from the CSV file *FILE*. This is identical to passing `--ratings-file=FILE` with `--delimiter=,`.

--tsv-file *FILE* Read ratings from the tab-separated file *FILE*. This is identical to passing `--ratings-file=FILE` with `--delimiter=^I`, but doesn't require you to know how to encode tab characters in your shell.

--pack-file *FILE* Read ratings from the packed rating file *FILE*. Packed files can be created with the **pack-ratings** command.

Additionally, the following options provide additional control over the data input:

-d *DELIM*, --delimiter *DELIM* Use *DELIM* as the delimiter for delimited text files. Only effective in conjunction with `--ratings-file`.

Script Environment Options

This command takes the standard LensKit script environment options:

-C *URL*, --classpath *URL* Add *URL* (which can be a path to a local directory or JAR file) to the classpath for loading the configuration scripts. This URL can contain additional components for the recommenders. This option can be specified multiple times to add multiple locations to the classpath.

-D *PROP=VALUE*, --define *PROP=VALUE* Define the property *PROP* to equal *VALUE*. This option is currently ignored for this command. To set Java system properties, use the `JAVA_OPTS` environment variable (see **lenskit**(1)).

See Also

lenskit(1)

A.4 lenskit-predict

Name

lenskit predict - predict user ratings of items.

Synopsis

lenskit [GLOBAL OPTIONS] **predict** [OPTIONS] *USER ITEM*...

Description

The `predict` command predicts a user's ratings for some items. It loads a recommender from a trained model file and/or LensKit configuration scripts and uses the configured algorithm to produce rating predictions.

Options

USER The user ID for whom to generate predictions.

ITEM An item ID to predict for.

--help Show usage help.

-m *FILE*, --model-file *FILE* Load a trained recommender engine from *FILE*.

-c *SCRIPT*, --config-file *SCRIPT* Configure the recommender using *SCRIPT*. This option can be specified multiple times, and later configurations take precedence over earlier ones. If `--model-file` is also specified, the scripts are used to modify the trained model.

--print-channel *CHAN* In addition to rating predictions, also print the value in side channel *CHAN*.

Input Data Options

This command can read data in several different ways. To give the rating prediction process some data to work with, one of the following mutually-exclusive options must be present:

--ratings-file *FILE* Read ratings from the delimited text file *FILE*.

--csv-file *FILE* Read ratings from the CSV file *FILE*. This is identical to passing `--ratings-file=FILE` with `--delimiter=,`.

--tsv-file *FILE* Read ratings from the tab-separated file *FILE*. This is identical to passing `--ratings-file=FILE` with `--delimiter=^I`, but doesn't require you to know how to encode tab characters in your shell.

--pack-file *FILE* Read ratings from the packed rating file *FILE*. Packed files can be created with the **pack-ratings** command.

Additionally, the following options provide additional control over the data input:

-d *DELIM*, --delimiter *DELIM* Use *DELIM* as the delimiter for delimited text files. Only effective in conjunction with `--ratings-file`.

Script Environment Options

This command takes the standard LensKit script environment options for controlling how configuration scripts are interpreted:

-C *URL*, --classpath *URL* Add *URL* (which can be a path to a local directory or JAR file) to the classpath for loading the configuration scripts. This URL can contain additional components for the recommenders. This option can be specified multiple times to add multiple locations to the classpath.

-D *PROP=VALUE*, --define *PROP=VALUE* Define the property *PROP* to equal *VALUE*. This option is currently ignored for this command. To set Java system properties, use the `JAVA_OPTS` environment variable (see **lenskit**(1)).

See Also

lenskit(1)

A.5 lenskit-recommend

Name

lenskit recommend - recommend items for users.

Synopsis

lenskit [GLOBAL OPTIONS] **recommend** [OPTIONS] *USER*...

Description

The `recommend` command recommends items for some users. It loads a recommender from a trained model file and/or LensKit configuration scripts and uses the configured algorithm to produce recommendations.

Options

USER A user to recommend for.

--help Show usage help.

-n *N* Produce *N* recommendations. The default is 10.

-m *FILE*, --model-file *FILE* Load a trained recommender engine from *FILE*.

-c *SCRIPT*, --config-file *SCRIPT* Configure the recommender using *SCRIPT*. This option can be specified multiple times, and later configurations take precedence over earlier ones. If `--model-file` is also specified, the scripts are used to modify the trained model.

--print-channel *CHAN* In addition to item scores, also print the value in side channel *CHAN*.

Input Data Options

This command can read data in several different ways. To give the recommendation process some data to work with, one of the following mutually-exclusive options must be present:

--ratings-file *FILE* Read ratings from the delimited text file *FILE*.

--csv-file *FILE* Read ratings from the CSV file *FILE*. This is identical to passing `--ratings-file=FILE` with `--delimiter=,`.

--tsv-file *FILE* Read ratings from the tab-separated file *FILE*. This is identical to passing `--ratings-file=FILE` with `--delimiter=^I`, but doesn't require you to know how to encode tab characters in your shell.

--pack-file *FILE* Read ratings from the packed rating file *FILE*. Packed files can be created with the **pack-ratings** command.

Additionally, the following options provide additional control over the data input:

-d *DELIM*, --delimiter *DELIM* Use *DELIM* as the delimiter for delimited text files. Only effective in conjunction with `--ratings-file`.

Script Environment Options

This command takes the standard LensKit script environment options for controlling how configuration scripts are interpreted:

-C *URL*, --classpath *URL* Add *URL* (which can be a path to a local directory or JAR file) to the classpath for loading the configuration scripts. This URL can contain additional components for the recommenders. This option can be specified multiple times to add multiple locations to the classpath.

-D *PROP=VALUE*, --define *PROP=VALUE* Define the property *PROP* to equal *VALUE*. This option is currently ignored for this command. To set Java system properties, use the `JAVA_OPTS` environment variable (see **lenskit**(1)).

See Also

lenskit(1)

A.6 lenskit-graph

Name

lenskit graph - produce a GraphViz diagram of a recommender configuration.

Synopsis

lenskit [GLOBAL OPTIONS] **graph** [OPTIONS] *CONFIGS*

Description

The *graph* command loads a LensKit algorithm configuration from one or more configuration files and produces a visualization of the resulting object graph. This visualization is in GraphViz DOT format, suitable for rendering with *dot*(1). Visualizing recommender configurations is often ueful for debugging configurations and making sure they produce the objects you expect.

Options

CONFIG A Groovy script containing a LensKit algorithm file in the LensKit configuration DSL. If there are multiple configurations, the are passed in order to `LenskitRecommenderEngineBuilder`, so later configurations override earlier ones.

--help Print usage help.

-o *FILE*, --output-file *FILE* Write the GraphViz file to *FILE*. The default output file is `recommender.dot`.

--domain *SPEC* Use the preference domain *SPEC* as the preference domain in the configuration. *SPEC* is of the form [*LOW*,*HIGH*]/*PREC*; the precision (and slash) can be omitted for continuously valued ratings. As an example, '[0.5,5.0]/0.5' will be a domain from 0.5 to 5.0 stars with a granularity of 1/2 star.

--model-file *FILE* Load a pre-trained model from *FILE*. In this mode, the configurations are applied as modifications to the model rather than used to build a graph from scratch. The mdoel file can be compressed.

Script Environment Options

This command takes the standard LensKit script environment options for controlling how configuration scripts are interpreted:

-C *URL*, --classpath *URL* Add *URL* (which can be a path to a local directory or JAR file) to the classpath for loading the configuration scripts. This URL can contain additional components for the recommenders. This option can be specified multiple times to add multiple locations to the classpath.

-D *PROP=VALUE*, --define *PROP=VALUE* Define the property *PROP* to equal *VALUE*. This option is currently ignored for this command. To set Java system properties, use the `JAVA_OPTS` environment variable (see **lenskit**(1)).

See Also

lenskit(1)

A.7 lenskit-eval

Name

lenskit eval - run an offline evaluation of recommender behavior and performance.

Synopsis

lenskit [GLOBAL OPTIONS] **eval** [OPTIONS] [*TARGET*...]

Description

The `eval` command runs a LensKit evaluation script to measure the behavior and performance (such as recommendation or prediction accuracy) of one or more recommender algorithms.

Evaluation scripts are written in Groovy, using an embedded domain-specific language for describing LensKit evaluations. This is documented more in the LensKit manual; there is a link in See Also.

The **lenskit eval** subcommand serves the same purpose as the now-deprecated **lenskit-eval** command, with slightly different invocation syntax. Use **lenskit eval** in new scripts and experiments.

Options

TARGET Run the target *TARGET* in the evaluation script. If no targets are specified on the command line, the script is run (which is sufficient for scripts that do not use targets), or the default target specified by the script is run.

--help Show usage help.

-f *SCRIPT*, --file *SCRIPT* Load evaluation script *SCRIPT*. The default is `eval.groovy`.

-j *N*, --thread-count *N* Use up to *N* threads for parallelizable portions of the evaluation.

-F, --force Force eval tasks to re-run, even if they detect that their outputs are up-to-date. Not all tasks do up-to-date checking.

Script Environment Options

This command takes the standard LensKit script environment options for controlling how configuration scripts are interpreted:

-C *URL*, --classpath *URL* Add *URL* (which can be a path to a local directory or JAR file) to the classpath for loading the evaluation script. This URL can contain additional components for the recommenders or evaluation. This option can be specified multiple times to add multiple locations to the classpath.

-D *PROP*=*VALUE*, --define *PROP*=*VALUE* Define the property *PROP* to equal *VALUE*. These properties are not Java system properties, but are available via the `config` object in evaluation scripts. This object can be accessed as a hash in Groovy.

See Also

- **lenskit**(1)
- Using the LensKit Evaluator

A.8 lenskit-pack-ratings

Name

lenskit pack-ratings - pack rating data into a binary file for efficient access.

Synopsis

lenskit [GLOBAL OPTIONS] **pack-ratings** *OPTIONS*

Description

The `pack-ratings` command packs rating data into a binary file that LensKit can efficiently map into memory. These files make many recommender operations significantly faster and less memory intensive, including model building and recommendation with certain algorithms.

Options

--help Show usage help.

-o *FILE*, --output-file *FILE* Write the resulting recommender model to *FILE*. If not specified, the ratings will be packed into the file `ratings.pack`.

--no-timestamps Ignore timestamps in the input data and omit them from the packed ratings.

Input Data Options

This command can read data in several different ways. One of the following mutually-exclusive options must be present:

--ratings-file *FILE* Read ratings from the delimited text file *FILE*.

--csv-file *FILE* Read ratings from the CSV file *FILE*. This is identical to passing `--ratings-file=FILE` with `--delimiter=,`.

--tsv-file *FILE* Read ratings from the tab-separated file *FILE*. This is identical to passing `--ratings-file=FILE` with `--delimiter=^I`, but doesn't require you to know how to encode tab characters in your shell.

--pack-file *FILE* Read ratings from the packed rating file *FILE*. Packed files can be created with the **pack-ratings** command.

Additionally, the following options provide additional control over the data input:

-d *DELIM*, --delimiter *DELIM* Use *DELIM* as the delimiter for delimited text files. Only effective in conjunction with `--ratings-file`.

Known Issues

If you want timestamped data, the input data must be sorted by timestamp. LensKit will eventually be able to sort data in the packing process, but cannot currently do so.

See Also

lenskit(1)

Appendix B

List Comparison SEM Output

This appendix contains the Lavaan output for the CFA and SEM models in chapter 7.

```
> library(lavaan)
> # load the compiled models
> load("experiment/build/models.Rdata")
```

B.1 Confirmatory Factor Analysis

```
> cat(cfa.full$spec)
```

```
Acc =~ NA * AccAppealing + AccAtTop + AccBad + AccBest
Sat =~ NA * SatFind + SatMobile + SatRecommend + SatSat + SatValuable
Und =~ NA * UndPersonalized + UndTaste + UndTrust + UndMainstream
Div =~ NA * DivMoods + DivSimilar + DivTastes + DivVaried
Nov =~ NA * NovFamiliar + NovFewerNew + NovSurprising + NovUnexpected + NovUnthought
Acc ~~ 1*Acc
Div ~~ 1*Div
Nov ~~ 1*Nov
Sat ~~ 1*Sat
Und ~~ 1*Und
Acc ~ CondSVDUU + CondIIUU
Div ~ CondSVDUU + CondIIUU
Nov ~ CondSVDUU + CondIIUU
Sat ~ CondSVDUU + CondIIUU
Und ~ CondSVDUU + CondIIUU
```

```
> summary(cfa.full$model)
```

```
lavaan (0.5-16) converged normally after  85 iterations

  Number of observations                           582

  Estimator                                       DWLS      Robust
```

```
  Minimum Function Test Statistic             1297.588    1508.457
  Degrees of freedom                               233         233
  P-value (Chi-square)                           0.000       0.000
  Scaling correction factor                                  0.932
  Shift parameter                                          116.428
    for simple second-order correction (Mplus variant)

Parameter estimates:

  Information                                 Expected
  Standard Errors                           Robust.sem

                   Estimate  Std.err  Z-value  P(>|z|)
Latent variables:
  Acc =~
    AccAppealing      0.911    0.010   86.949    0.000
    AccAtTop          0.572    0.027   21.074    0.000
    AccBad           -0.751    0.020  -38.429    0.000
    AccBest           0.786    0.016   48.888    0.000
  Sat =~
    SatFind           0.923    0.007  125.266    0.000
    SatMobile         0.921    0.008  112.673    0.000
    SatRecommend      0.846    0.012   69.073    0.000
    SatSat            0.928    0.007  129.929    0.000
    SatValuable       0.884    0.010   91.959    0.000
  Und =~
    UndPersonalzd     0.842    0.012   68.642    0.000
    UndTaste          0.933    0.007  138.226    0.000
    UndTrust          0.943    0.007  144.785    0.000
    UndMainstream    -0.072    0.036   -2.010    0.044
  Div =~
    DivMoods          0.838    0.015   57.801    0.000
    DivSimilar       -0.772    0.019  -41.423    0.000
    DivTastes         0.793    0.017   46.592    0.000
    DivVaried         0.772    0.018   42.440    0.000
  Nov =~
    NovFamiliar      -0.784    0.023  -34.118    0.000
    NovFewerNew      -0.258    0.037   -7.011    0.000
    NovSurprising    -0.454    0.038  -11.931    0.000
    NovUnexpected     0.770    0.023   33.509    0.000
```

```
    NovUnthought      0.704    0.025   28.172    0.000

Regressions:
  Acc ~
    CondSVDUU        -0.774    0.119   -6.502    0.000
    CondIIUU         -0.709    0.111   -6.417    0.000
  Div ~
    CondSVDUU         0.601    0.119    5.066    0.000
    CondIIUU          0.295    0.111    2.664    0.008
  Nov ~
    CondSVDUU         0.961    0.120    7.998    0.000
    CondIIUU          0.912    0.113    8.043    0.000
  Sat ~
    CondSVDUU        -0.592    0.113   -5.249    0.000
    CondIIUU         -0.548    0.105   -5.232    0.000
  Und ~
    CondSVDUU        -0.652    0.117   -5.577    0.000
    CondIIUU         -0.534    0.107   -4.997    0.000

Covariances:
  Acc ~~
    Sat               0.927    0.009  107.165    0.000
    Und               0.989    0.008  127.071    0.000
    Div              -0.033    0.041   -0.801    0.423
    Nov              -0.775    0.022  -34.995    0.000
  Sat ~~
    Und               0.966    0.005  198.037    0.000
    Div               0.125    0.038    3.278    0.001
    Nov              -0.626    0.027  -23.174    0.000
  Und ~~
    Div               0.090    0.039    2.332    0.020
    Nov              -0.676    0.026  -26.391    0.000
  Div ~~
    Nov               0.090    0.040    2.279    0.023

Intercepts:
    Acc               0.000
    Sat               0.000
    Und               0.000
    Div               0.000
```

```
    Nov               0.000

Thresholds:
    AccAppelng|t1    -1.377    0.097  -14.225    0.000
    AccAppelng|t2    -0.265    0.084   -3.148    0.002
    AccAppelng|t3     0.303    0.085    3.584    0.000
    AccAppelng|t4     1.450    0.117   12.370    0.000
    AccAtTop|t1      -1.439    0.093  -15.412    0.000
    AccAtTop|t2      -0.517    0.081   -6.361    0.000
    AccAtTop|t3       0.415    0.081    5.122    0.000
    AccAtTop|t4       1.316    0.100   13.100    0.000
    AccBad|t1        -1.385    0.109  -12.674    0.000
    AccBad|t2        -0.495    0.086   -5.752    0.000
    AccBad|t3         0.567    0.087    6.548    0.000
    AccBad|t4         1.589    0.100   15.846    0.000
    AccBest|t1       -1.442    0.094  -15.338    0.000
    AccBest|t2       -0.535    0.082   -6.532    0.000
    AccBest|t3        0.671    0.083    8.045    0.000
    AccBest|t4        1.603    0.117   13.645    0.000
    SatFind|t1       -1.564    0.099  -15.863    0.000
    SatFind|t2       -0.456    0.084   -5.446    0.000
    SatFind|t3        0.515    0.084    6.094    0.000
    SatFind|t4        1.659    0.125   13.243    0.000
    SatMobile|t1     -1.606    0.105  -15.292    0.000
    SatMobile|t2     -0.593    0.088   -6.707    0.000
    SatMobile|t3      0.685    0.090    7.646    0.000
    SatMobile|t4      1.715    0.131   13.089    0.000
    SatRecmmnd|t1    -1.673    0.098  -17.092    0.000
    SatRecmmnd|t2    -0.631    0.085   -7.447    0.000
    SatRecmmnd|t3     0.731    0.086    8.481    0.000
    SatRecmmnd|t4     1.832    0.138   13.244    0.000
    SatSat|t1        -1.631    0.096  -16.943    0.000
    SatSat|t2        -0.416    0.083   -5.039    0.000
    SatSat|t3         0.611    0.085    7.215    0.000
    SatSat|t4         1.726    0.130   13.307    0.000
    SatValuabl|t1    -1.601    0.099  -16.130    0.000
    SatValuabl|t2    -0.493    0.086   -5.746    0.000
    SatValuabl|t3     0.501    0.086    5.824    0.000
    SatValuabl|t4     1.578    0.116   13.596    0.000
    UndPrsnlzd|t1    -1.604    0.102  -15.708    0.000
```

UndPrsnlzd|t2	-0.446	0.085	-5.239	0.000
UndPrsnlzd|t3	0.475	0.085	5.564	0.000
UndPrsnlzd|t4	1.687	0.127	13.266	0.000
UndTaste|t1	-1.432	0.098	-14.614	0.000
UndTaste|t2	-0.337	0.085	-3.955	0.000
UndTaste|t3	0.576	0.088	6.565	0.000
UndTaste|t4	1.493	0.120	12.422	0.000
UndTrust|t1	-1.522	0.096	-15.891	0.000
UndTrust|t2	-0.356	0.083	-4.292	0.000
UndTrust|t3	0.562	0.085	6.620	0.000
UndTrust|t4	1.498	0.116	12.891	0.000
UndManstrm|t1	-1.563	0.103	-15.155	0.000
UndManstrm|t2	-0.588	0.082	-7.184	0.000
UndManstrm|t3	0.665	0.083	8.051	0.000
UndManstrm|t4	1.566	0.108	14.476	0.000
DivMoods|t1	-1.444	0.115	-12.508	0.000
DivMoods|t2	-0.266	0.087	-3.072	0.002
DivMoods|t3	0.732	0.091	8.080	0.000
DivMoods|t4	1.818	0.119	15.322	0.000
DivSimilar|t1	-1.596	0.108	-14.765	0.000
DivSimilar|t2	-0.673	0.085	-7.931	0.000
DivSimilar|t3	0.245	0.083	2.968	0.003
DivSimilar|t4	1.433	0.107	13.332	0.000
DivTastes|t1	-1.383	0.110	-12.575	0.000
DivTastes|t2	-0.284	0.088	-3.230	0.001
DivTastes|t3	0.600	0.091	6.608	0.000
DivTastes|t4	1.754	0.121	14.476	0.000
DivVaried|t1	-1.307	0.103	-12.733	0.000
DivVaried|t2	-0.277	0.083	-3.335	0.001
DivVaried|t3	0.695	0.086	8.091	0.000
DivVaried|t4	1.796	0.113	15.867	0.000
NovFamilir|t1	-1.410	0.096	-14.756	0.000
NovFamilir|t2	-0.429	0.084	-5.118	0.000
NovFamilir|t3	0.568	0.086	6.628	0.000
NovFamilir|t4	1.531	0.122	12.552	0.000
NovFewerNw|t1	-1.706	0.104	-16.370	0.000
NovFewerNw|t2	-0.744	0.089	-8.402	0.000
NovFewerNw|t3	0.594	0.087	6.832	0.000
NovFewerNw|t4	1.432	0.115	12.446	0.000
NovSrprsng|t1	-1.771	0.113	-15.633	0.000

```
    NovSrprsng|t2    -0.610    0.086   -7.110    0.000
    NovSrprsng|t3     0.863    0.089    9.745    0.000
    NovSrprsng|t4     2.040    0.147   13.895    0.000
    NovUnxpctd|t1    -1.268    0.108  -11.731    0.000
    NovUnxpctd|t2    -0.422    0.084   -5.004    0.000
    NovUnxpctd|t3     0.435    0.084    5.175    0.000
    NovUnxpctd|t4     1.590    0.099   16.090    0.000
    NovUnthght|t1    -1.479    0.114  -12.992    0.000
    NovUnthght|t2    -0.542    0.081   -6.675    0.000
    NovUnthght|t3     0.669    0.081    8.221    0.000
    NovUnthght|t4     1.631    0.096   16.980    0.000

Variances:
    Acc               1.000
    Div               1.000
    Nov               1.000
    Sat               1.000
    Und               1.000
    AccAppealing      0.170
    AccAtTop          0.672
    AccBad            0.437
    AccBest           0.382
    SatFind           0.149
    SatMobile         0.152
    SatRecommend      0.285
    SatSat            0.138
    SatValuable       0.219
    UndPersonalzd     0.290
    UndTaste          0.129
    UndTrust          0.111
    UndMainstream     0.995
    DivMoods          0.297
    DivSimilar        0.405
    DivTastes         0.372
    DivVaried         0.405
    NovFamiliar       0.385
    NovFewerNew       0.934
    NovSurprising     0.794
    NovUnexpected     0.407
    NovUnthought      0.504
```

B.2 Overall SEM

```
> cat(sem.measure$spec)

Sat =~ SatFind + SatMobile + SatRecommend + SatSat + SatValuable
Div =~ DivMoods + DivSimilar + DivTastes + DivVaried
Nov =~ NovUnexpected + NovFamiliar + NovUnthought
PopRatio ~ CondIIUU + CondSVDUU
Div ~ Nov + SimRatio
Nov ~ CondIIUU + CondSVDUU + PopRatio
Sat ~ Nov + Div + PredAccRatio
FirstImpression ~ Sat + Nov
PickedB ~ Sat + FirstImpression

> summary(sem.measure$model)

lavaan (0.5-16) converged normally after  87 iterations

  Number of observations                           582

  Estimator                                       DWLS
  Minimum Function Test Statistic             229.520
  Degrees of freedom                              139
  P-value (Chi-square)                          0.000

Parameter estimates:

  Information                                 Observed
  Standard Errors                            Bootstrap
  Number of requested bootstrap draws             1000
  Number of successful bootstrap draws            1000

                   Estimate  Std.err  Z-value  P(>|z|)
Latent variables:
  Sat =~
    SatFind           0.737    0.029   25.714    0.000
    SatMobile         0.736    0.028   25.938    0.000
    SatRecommend      0.678    0.027   25.093    0.000
    SatSat            0.745    0.028   26.147    0.000
    SatValuable       0.717    0.029   24.497    0.000
  Div =~
```

```
    DivMoods          0.806    0.033   24.703    0.000
    DivSimilar       -0.748    0.033  -22.331    0.000
    DivTastes         0.768    0.031   24.819    0.000
    DivVaried         0.743    0.034   21.928    0.000
  Nov =~
    NovUnexpected     0.750    0.038   19.582    0.000
    NovFamiliar      -0.762    0.031  -24.359    0.000
    NovUnthought      0.707    0.036   19.373    0.000

Regressions:
  PopRatio ~
    CondIIUU          0.223    0.028    7.916    0.000
    CondSVDUU         0.189    0.028    6.670    0.000
  Div ~
    Nov               0.184    0.056    3.290    0.001
    SimRatio        -51.756    8.558   -6.047    0.000
  Nov ~
    CondIIUU          0.835    0.153    5.442    0.000
    CondSVDUU         1.042    0.149    7.005    0.000
    PopRatio          1.309    0.206    6.352    0.000
  Sat ~
    Nov              -0.700    0.073   -9.556    0.000
    Div               0.270    0.061    4.396    0.000
    PredAccRatio      1.057    0.509    2.078    0.038
  FirstImpression ~
    Sat               0.542    0.037   14.523    0.000
    Nov              -0.249    0.038   -6.496    0.000
  PickedB ~
    Sat               0.664    0.043   15.290    0.000
    FirstImpressn     0.093    0.031    2.983    0.003

Intercepts:
    PopRatio         -0.020    0.019   -1.064    0.287
    Sat               0.000
    Div               0.000
    Nov               0.000

Thresholds:
    SatFind|t1       -1.647    0.102  -16.147    0.000
    SatFind|t2       -0.528    0.083   -6.365    0.000
```

SatFind|t3	0.453	0.086	5.268	0.000
SatFind|t4	1.610	0.127	12.635	0.000
SatMobile|t1	-1.657	0.104	-15.939	0.000
SatMobile|t2	-0.639	0.088	-7.238	0.000
SatMobile|t3	0.649	0.089	7.313	0.000
SatMobile|t4	1.687	0.135	12.523	0.000
SatRecmmnd|t1	-1.769	0.107	-16.552	0.000
SatRecmmnd|t2	-0.717	0.090	-7.940	0.000
SatRecmmnd|t3	0.660	0.091	7.291	0.000
SatRecmmnd|t4	1.776	0.147	12.110	0.000
SatSat|t1	1.700	0.106	16.037	0.000
SatSat|t2	-0.454	0.088	-5.160	0.000
SatSat|t3	0.579	0.089	6.521	0.000
SatSat|t4	1.699	0.133	12.800	0.000
SatValuabl|t1	-1.642	0.104	-15.787	0.000
SatValuabl|t2	-0.526	0.086	-6.143	0.000
SatValuabl|t3	0.472	0.086	5.506	0.000
SatValuabl|t4	1.551	0.121	12.858	0.000
DivMoods|t1	-1.614	0.105	-15.407	0.000
DivMoods|t2	-0.414	0.078	-5.303	0.000
DivMoods|t3	0.603	0.080	7.587	0.000
DivMoods|t4	1.731	0.102	16.905	0.000
DivSimilar|t1	-1.497	0.089	-16.752	0.000
DivSimilar|t2	-0.548	0.082	-6.692	0.000
DivSimilar|t3	0.395	0.082	4.840	0.000
DivSimilar|t4	-1.595	0.114	-13.942	0.000
DivTastes|t1	-1.573	0.107	-14.760	0.000
DivTastes|t2	-0.451	0.080	-5.611	0.000
DivTastes|t3	0.458	0.077	5.926	0.000
DivTastes|t4	1.660	0.097	17.084	0.000
DivVaried|t1	-1.472	0.107	-13.746	0.000
DivVaried|t2	-0.425	0.082	-5.159	0.000
DivVaried|t3	0.572	0.084	6.833	0.000
DivVaried|t4	1.706	0.101	16.938	0.000
NovUnxpctd|t1	-1.269	0.106	-11.988	0.000
NovUnxpctd|t2	-0.423	0.086	-4.948	0.000
NovUnxpctd|t3	0.435	0.093	4.702	0.000
NovUnxpctd|t4	1.591	0.107	14.849	0.000
NovFamilir|t1	-1.477	0.106	-13.976	0.000
NovFamilir|t2	-0.491	0.093	-5.278	0.000

```
    NovFamilir|t3    -0.477    0.085   -5.592    0.000
    NovFamilir|t4     1.497    0.132   11.303    0.000
    NovUnthght|t1    -1.467    0.125  -11.742    0.000
    NovUnthght|t2    -0.531    0.099   -5.368    0.000
    NovUnthght|t3     0.685    0.104    6.573    0.000
    NovUnthght|t4     1.651    0.114   14.496    0.000
    FrstImprssn|1    -1.492    0.102  -14.607    0.000
    FrstImprssn|2    -0.429    0.086   -4.962    0.000
    FrstImprssn|3     0.374    0.090    4.160    0.000
    FrstImprssn|4     1.417    0.120   11.773    0.000
    PickedB|t1       -0.059    0.099   -0.596    0.551

Variances:
    SatFind           0.158
    SatMobile         0.160
    SatRecommend      0.288
    SatSat            0.140
    SatValuable       0.202
    DivMoods          0.325
    DivSimilar        0.419
    DivTastes         0.388
    DivVaried         0.426
    NovUnexpected     0.366
    NovFamiliar       0.344
    NovUnthought      0.436
    PopRatio          0.075    0.006
    FirstImpressn     0.276
    PickedB           0.183
    Sat               1.000
    Div               1.000
    Nov               1.000
```

B.3 Pseudo-experiment SEMs

SVD vs. User-User

```
> cat(split.models$svd.uu$spec)

Sat =~ 0.737164412922786 * SatFind
Sat =~ 0.736418615686128 * SatMobile
Sat =~ 0.67807484821091 * SatRecommend
```

```
Sat =~ 0.744863507100357 * SatSat
Sat =~ 0.717453826497938 * SatValuable
Sat ~~ 1 * Sat
Div =~ 0.806142608188462 * DivMoods
Div =~ -0.747928415795148 * DivSimilar
Div =~ 0.76779608551984 * DivTastes
Div =~ 0.743254221458815 * DivVaried
Div ~~ 1 * Div
Nov =~ 0.74962562141753 * NovUnexpected
Nov =~ -0.76228134082548 * NovFfamiliar
Nov =~ 0.706649983622846 * NovUnthought
Nov ~~ 1 * Nov
Div ~ CondIIUU # Novelty not significant
Nov ~ CondIIUU
Sat ~ Nov + Div
FirstImpression ~ Sat + Nov
PickedB ~ Sat

> summary(split.models$svd.uu$model)

lavaan (0.5-16) converged normally after  28 iterations

  Number of observations                           399

  Estimator                                       DWLS
  Minimum Function Test Statistic             200.840
  Degrees of freedom                               97
  P-value (Chi-square)                          0.000

Parameter estimates:

  Information                                 Observed
  Standard Errors                            Bootstrap
  Number of requested bootstrap draws             1000
  Number of successful bootstrap draws             996

                   Estimate  Std.err  Z-value  P(>|z|)
Latent variables:
  Sat =~
    SatFind           0.737
    SatMobile         0.736
```

```
    SatRecommend      0.678
    SatSat            0.745
    SatValuable       0.717
  Div =~
    DivMoods          0.806
    DivSimilar       -0.748
    DivTastes         0.768
    DivVaried         0.743
  Nov =~
    NovUnexpected     0.750
    NovFamiliar      -0.762
    NovUnthought      0.707

Regressions:
  Div ~
    CondIIUU          0.312    0.111    2.821    0.005
  Nov ~
    CondIIUU          0.953    0.136    6.994    0.000
  Sat ~
    Nov              -0.759    0.023  -32.636    0.000
    Div               0.129    0.078    1.642    0.101
  FirstImpression ~
    Sat               0.514    0.047   10.944    0.000
    Nov              -0.302    0.077   -3.912    0.000
  PickedB ~
    Sat               0.730    0.025   29.218    0.000

Covariances:
  FirstImpression ~~
    PickedB           0.000    0.040    0.000    1.000

Intercepts:
    Sat               0.000
    Div               0.000
    Nov               0.000

Thresholds:
    SatFind|t1       -1.496    0.101  -14.752    0.000
    SatFind|t2       -0.452    0.083   -5.452    0.000
    SatFind|t3        0.492    0.082    5.973    0.000
```

SatFind|t4	1.641	0.137	11.975	0.000
SatMobile|t1	-1.535	0.099	-15.450	0.000
SatMobile|t2	-0.598	0.083	-7.195	0.000
SatMobile|t3	0.691	0.083	8.373	0.000
SatMobile|t4	1.631	0.138	11.857	0.000
SatRecmmnd|t1	-1.543	0.109	-14.183	0.000
SatRecmmnd|t2	-0.635	0.086	-7.360	0.000
SatRecmmnd|t3	0.701	0.086	8.112	0.000
SatRecmmnd|t4	1.807	0.150	12.083	0.000
SatSat|t1	-1.549	0.113	-13.693	0.000
SatSat|t2	-0.412	0.084	-4.898	0.000
SatSat|t3	0.580	0.085	6.806	0.000
SatSat|t4	1.719	0.147	11.672	0.000
SatValuabl|t1	-1.529	0.107	-14.268	0.000
SatValuabl|t2	-0.527	0.083	-6.329	0.000
SatValuabl|t3	0.525	0.078	6.699	0.000
SatValuabl|t4	1.535	0.124	12.408	0.000
DivMoods|t1	-1.473	0.108	-13.683	0.000
DivMoods|t2	-0.246	0.079	-3.103	0.002
DivMoods|t3	0.700	0.081	8.684	0.000
DivMoods|t4	1.903	0.129	14.731	0.000
DivSimilar|t1	-1.631	0.103	-15.784	0.000
DivSimilar|t2	-0.647	0.086	-7.548	0.000
DivSimilar|t3	0.217	0.081	2.678	0.007
DivSimilar|t4	1.470	0.112	13.163	0.000
DivTastes|t1	-1.391	0.105	-13.245	0.000
DivTastes|t2	-0.288	0.077	-3.723	0.000
DivTastes|t3	0.606	0.079	7.634	0.000
DivTastes|t4	1.765	0.111	15.912	0.000
DivVaried|t1	-1.300	0.102	-12.774	0.000
DivVaried|t2	-0.268	0.078	-3.419	0.001
DivVaried|t3	0.673	0.084	8.012	0.000
DivVaried|t4	1.822	0.125	14.560	0.000
NovUnxpctd|t1	-1.221	0.106	-11.466	0.000
NovUnxpctd|t2	-0.406	0.080	-5.055	0.000
NovUnxpctd|t3	0.426	0.084	5.095	0.000
NovUnxpctd|t4	1.511	0.104	14.597	0.000
NovFamilir|t1	-1.346	0.090	-14.912	0.000
NovFamilir|t2	-0.435	0.084	-5.157	0.000
NovFamilir|t3	0.543	0.084	6.454	0.000

```
    NovFamilir|t4     1.536    0.131   11.726    0.000
    NovUnthght|t1    -1.438    0.120  -12.000    0.000
    NovUnthght|t2    -0.479    0.088   -5.431    0.000
    NovUnthght|t3     0.611    0.094    6.521    0.000
    NovUnthght|t4     1.557    0.106   14.730    0.000
    FrstImprssn|1    -1.357    0.098  -13.855    0.000
    FrstImprssn|2    -0.364    0.080   -4.562    0.000
    FrstImprssn|3     0.457    0.082    5.578    0.000
    FrstImprssn|4     1.406    0.117   12.007    0.000
    PickedB|t1        0.025    0.090    0.281    0.779

Variances:
    Sat               1.000
    Div               1.000
    Nov               1.000
    SatFind           0.135
    SatMobile         0.136
    SatRecommend      0.268
    SatSat            0.117
    SatValuable       0.180
    DivMoods          0.350
    DivSimilar        0.441
    DivTastes         0.410
    DivVaried         0.448
    NovUnexpected     0.438
    NovFamiliar       0.419
    NovUnthought      0.501
    FirstImpressn     0.254
    PickedB           0.152
```

Item-Item vs. User-User

```
> cat(split.models$uu.ii$spec)

Sat =~ 0.737164412922786 * SatFind
Sat =~ 0.736418615686128 * SatMobile
Sat =~ 0.67807484821091 * SatRecommend
Sat =~ 0.744863507100357 * SatSat
Sat =~ 0.717453826497938 * SatValuable
Sat ~~ 1 * Sat
Div =~ 0.806142608188462 * DivMoods
```

```
Div =~ -0.747928415795148 * DivSimilar
Div =~ 0.76779608551984 * DivTastes
Div =~ 0.743254221458815 * DivVaried
Div ~~ 1 * Div
Nov =~ 0.74962562141753 * NovUnexpected
Nov =~ -0.76228134082548 * NovFamiliar
Nov =~ 0.706649983622846 * NovUnthought
Nov ~~ 1 * Nov
Div ~ Nov
Nov ~ CondIISVD + HighRatings + CondHighRatings
Sat ~ Nov + Div
FirstImpression ~ Sat + Nov
PickedB ~ Sat
```

```
> summary(split.models$uu.ii$model)

lavaan (0.5-16) converged normally after  34 iterations

  Number of observations                           381

  Estimator                                       DWLS
  Minimum Function Test Statistic              177.453
  Degrees of freedom                               123
  P-value (Chi-square)                           0.001

Parameter estimates:

  Information                                 Observed
  Standard Errors                            Bootstrap
  Number of requested bootstrap draws             1000
  Number of successful bootstrap draws             999

                   Estimate  Std.err  Z-value  P(>|z|)
Latent variables:
  Sat =~
    SatFind           0.737
    SatMobile         0.736
    SatRecommend      0.678
    SatSat            0.745
    SatValuable       0.717
  Div =~
```

```
    DivMoods          0.806
    DivSimilar       -0.748
    DivTastes         0.768
    DivVaried         0.743
  Nov =~
    NovUnexpected     0.750
    NovFamiliar      -0.762
    NovUnthought      0.707

Regressions:
  Div ~
    Nov               0.200    0.081    2.475    0.013
  Nov ~
    CondIISVD        -1.563    0.207   -7.561    0.000
    HighRatings      -0.459    0.185   -2.482    0.013
    CondHighRtngs     0.712    0.258    2.763    0.006
  Sat ~
    Nov              -0.699    0.040  -17.490    0.000
    Div               0.391    0.069    5.660    0.000
  FirstImpression ~
    Sat               0.525    0.037   14.307    0.000
    Nov              -0.304    0.060   -5.101    0.000
  PickedB ~
    Sat               0.737    0.023   32.026    0.000

Covariances:
  FirstImpression ~~
    PickedB           0.000    0.034    0.000    1.000

Intercepts:
    Sat               0.000
    Div               0.000
    Nov               0.000

Thresholds:
    SatFind|t1       -1.022    0.129   -7.914    0.000
    SatFind|t2        0.199    0.119    1.669    0.095
    SatFind|t3        1.246    0.131    9.519    0.000
    SatFind|t4        2.359    0.172   13.741    0.000
    SatMobile|t1     -1.071    0.137   -7.837    0.000
```

SatMobile|t2	0.013	0.128	0.104	0.917
SatMobile|t3	1.388	0.145	9.592	0.000
SatMobile|t4	2.594	0.200	12.957	0.000
SatRecmmnd|t1	-1.174	0.130	-9.051	0.000
SatRecmmnd|t2	0.042	0.115	0.364	0.716
SatRecmmnd|t3	1.490	0.136	10.966	0.000
SatRecmmnd|t4	2.438	0.167	14.605	0.000
SatSat|t1	-1.116	0.130	-8.566	0.000
SatSat|t2	0.190	0.119	1.593	0.111
SatSat|t3	1.316	0.135	9.775	0.000
SatSat|t4	2.381	0.162	14.659	0.000
SatValuabl|t1	-1.131	0.127	-8.933	0.000
SatValuabl|t2	0.044	0.121	0.369	0.712
SatValuabl|t3	1.114	0.132	8.448	0.000
SatValuabl|t4	2.210	0.160	13.856	0.000
DivMoods|t1	-1.906	0.169	-11.309	0.000
DivMoods|t2	-0.787	0.124	-6.348	0.000
DivMoods|t3	0.318	0.122	2.616	0.009
DivMoods|t4	1.462	0.139	10.506	0.000
DivSimilar|t1	-0.923	0.127	-7.276	0.000
DivSimilar|t2	0.094	0.115	0.812	0.417
DivSimilar|t3	1.045	0.118	8.837	0.000
DivSimilar|t4	2.193	0.179	12.244	0.000
DivTastes|t1	-2.086	0.199	-10.492	0.000
DivTastes|t2	-0.832	0.125	-6.672	0.000
DivTastes|t3	0.127	0.118	1.077	0.281
DivTastes|t4	1.294	0.136	9.537	0.000
DivVaried|t1	-2.160	0.181	-11.931	0.000
DivVaried|t2	-1.013	0.120	-8.409	0.000
DivVaried|t3	0.017	0.114	0.151	0.880
DivVaried|t4	1.103	0.127	8.655	0.000
NovUnxpctd|t1	-2.623	0.184	-14.288	0.000
NovUnxpctd|t2	-1.460	0.135	-10.840	0.000
NovUnxpctd|t3	-0.547	0.120	-4.553	0.000
NovUnxpctd|t4	0.598	0.118	5.080	0.000
NovFamilir|t1	-0.431	0.121	-3.574	0.000
NovFamilir|t2	0.670	0.127	5.291	0.000
NovFamilir|t3	1.722	0.144	11.953	0.000
NovFamilir|t4	2.722	0.177	15.402	0.000
NovUnthght|t1	-2.390	0.184	-13.011	0.000

```
    NovUnthght|t2    -1.415    0.126  -11.206    0.000
    NovUnthght|t3    -0.182    0.116   -1.568    0.117
    NovUnthght|t4     0.806    0.126    6.389    0.000
    FrstImprssn|1    -0.621    0.123   -5.069    0.000
    FrstImprssn|2     0.413    0.123    3.344    0.001
    FrstImprssn|3     1.294    0.144    8.989    0.000
    FrstImprssn|4     2.401    0.163   14.697    0.000
    PickedB|t1        0.658    0.143    4.592    0.000

Variances:
    Sat               1.000
    Div               1.000
    Nov               1.000
    SatFind           0.164
    SatMobile         0.166
    SatRecommend      0.293
    SatSat            0.147
    SatValuable       0.208
    DivMoods          0.324
    DivSimilar        0.418
    DivTastes         0.387
    DivVaried         0.425
    NovUnexpected     0.438
    NovFamiliar       0.419
    NovUnthought      0.501
    FirstImpressn     0.286
    PickedB           0.165
```

Item-Item vs. SVD

```
> cat(split.models$ii.svd$spec)

Sat =~ 0.737164412922786 * SatFind
Sat =~ 0.736418615686128 * SatMobile
Sat =~ 0.67807484821091 * SatRecommend
Sat =~ 0.744863507100357 * SatSat
Sat =~ 0.717453826497938 * SatValuable
Sat ~~ 1 * Sat
Div =~ 0.806142608188462 * DivMoods
Div =~ -0.747928415795148 * DivSimilar
Div =~ 0.76779608551984 * DivTastes
```

```
Div =~ 0.743254221458815 * DivVaried
Div ~~ 1 * Div
Nov =~ 0.74962562141753 * NovUnexpected
Nov =~ -0.76228134082548 * NovFamiliar
Nov =~ 0.706649983622846 * NovUnthought
Nov ~~ 1 * Nov
Div ~ CondSVDUU + Nov
Nov ~ HighRatings
Sat ~ Nov + Div
FirstImpression ~ Sat + Nov
PickedB ~ Sat

> summary(split.models$ii.svd$model)

lavaan (0.5-16) converged normally after  28 iterations

  Number of observations                           384

  Estimator                                       DWLS
  Minimum Function Test Statistic             167.004
  Degrees of freedom                              110
  P-value (Chi-square)                          0.000

Parameter estimates:

  Information                                 Observed
  Standard Errors                            Bootstrap
  Number of requested bootstrap draws             1000
  Number of successful bootstrap draws             924

                   Estimate  Std.err  Z-value  P(>|z|)
Latent variables:
  Sat =~
    SatFind           0.737
    SatMobile         0.736
    SatRecommend      0.678
    SatSat            0.745
    SatValuable       0.717
  Div =~
    DivMoods          0.806
    DivSimilar       -0.748
```

```
    DivTastes         0.768
    DivVaried         0.743
  Nov =~
    NovUnexpected     0.750
    NovFamiliar      -0.762
    NovUnthought      0.707

Regressions:
  Div ~
    CondSVDUU        -0.260    0.118   -2.200    0.028
    Nov               0.215    0.084    2.568    0.010
  Nov ~
    HighRatings       0.570    0.129    4.427    0.000
  Sat ~
    Nov              -0.747    0.032  -23.119    0.000
    Div               0.254    0.075    3.371    0.001
  FirstImpression ~
    Sat               0.533    0.043   12.278    0.000
    Nov              -0.303    0.073   -4.134    0.000
  PickedB ~
    Sat               0.714    0.031   23.060    0.000

Covariances:
  FirstImpression ~~
    PickedB           0.000    0.041    0.000    1.000

Intercepts:
    Sat               0.000
    Div               0.000
    Nov               0.000

Thresholds:
    SatFind|t1       -2.224    0.189  -11.757    0.000
    SatFind|t2       -1.137    0.120   -9.474    0.000
    SatFind|t3       -0.203    0.103   -1.967    0.049
    SatFind|t4        0.925    0.109    8.449    0.000
    SatMobile|t1     -2.358    0.216  -10.909    0.000
    SatMobile|t2     -1.252    0.119  -10.551    0.000
    SatMobile|t3     -0.078    0.100   -0.785    0.433
    SatMobile|t4      0.935    0.111    8.385    0.000
```

SatRecmmnd|t1	-2.630	0.214	-12.308	0.000
SatRecmmnd|t2	-1.432	0.125	-11.441	0.000
SatRecmmnd|t3	-0.100	0.105	-0.947	0.344
SatRecmmnd|t4	1.002	0.115	8.691	0.000
SatSat|t1	-2.312	0.194	-11.921	0.000
SatSat|t2	-1.252	0.121	-10.333	0.000
SatSat|t3	-0.271	0.104	-2.605	0.009
SatSat|t4	1.012	0.114	8.846	0.000
SatValuabl|t1	-2.162	0.173	-12.469	0.000
SatValuabl|t2	-0.995	0.116	-8.545	0.000
SatValuabl|t3	-0.131	0.105	-1.244	0.213
SatValuabl|t4	1.039	0.114	9.129	0.000
DivMoods|t1	-1.469	0.119	-12.372	0.000
DivMoods|t2	-0.409	0.103	-3.985	0.000
DivMoods|t3	0.540	0.105	5.159	0.000
DivMoods|t4	1.585	0.135	11.784	0.000
DivSimilar|t1	-1.852	0.159	-11.651	0.000
DivSimilar|t2	-0.697	0.108	-6.454	0.000
DivSimilar|t3	0.263	0.106	2.479	0.013
DivSimilar|t4	1.097	0.112	9.786	0.000
DivTastes|t1	-1.457	0.120	-12.146	0.000
DivTastes|t2	-0.364	0.098	-3.700	0.000
DivTastes|t3	0.439	0.101	4.326	0.000
DivTastes|t4	1.487	0.132	11.240	0.000
DivVaried|t1	-1.491	0.121	-12.369	0.000
DivVaried|t2	-0.424	0.099	-4.289	0.000
DivVaried|t3	0.540	0.102	5.314	0.000
DivVaried|t4	1.512	0.136	11.150	0.000
NovUnxpctd|t1	-0.676	0.106	-6.364	0.000
NovUnxpctd|t2	0.502	0.101	4.998	0.000
NovUnxpctd|t3	1.340	0.116	11.528	0.000
NovUnxpctd|t4	2.076	0.164	12.635	0.000
NovFamilir|t1	-2.581	0.201	-12.833	0.000
NovFamilir|t2	-1.664	0.142	-11.677	0.000
NovFamilir|t3	-0.679	0.115	-5.910	0.000
NovFamilir|t4	0.332	0.112	2.973	0.003
NovUnthght|t1	-0.730	0.104	-7.053	0.000
NovUnthght|t2	0.222	0.103	2.151	0.031
NovUnthght|t3	1.581	0.127	12.498	0.000
NovUnthght|t4	2.473	0.228	10.824	0.000

```
    FrstImprssn|1    -2.492    0.208  -11.994    0.000
    FrstImprssn|2    -1.204    0.121   -9.937    0.000
    FrstImprssn|3    -0.540    0.105   -5.124    0.000
    FrstImprssn|4     0.570    0.107    5.315    0.000
    PickedB|t1       -0.681    0.115   -5.941    0.000

Variances:
    Sat               1.000
    Div               1.000
    Nov               1.000
    SatFind           0.161
    SatMobile         0.162
    SatRecommend      0.290
    SatSat            0.143
    SatValuable       0.205
    DivMoods          0.320
    DivSimilar        0.415
    DivTastes         0.383
    DivVaried         0.422
    NovUnexpected     0.438
    NovFamiliar       0.419
    NovUnthought      0.501
    FirstImpressn     0.247
    PickedB           0.214
```