Rethinking the Recommender Research Ecosystem: Reproducibility, Openness, and LensKit

Michael D. Ekstrand, Michael Ludwig, Joseph A. Konstan, and John T. Riedl
GroupLens Research
University of Minnesota
Department of Computer Science
{ekstrand,mludwig,konstan,riedl}@cs.umn.edu

ABSTRACT

Recommender systems research is being slowed by the difficulty of replicating and comparing research results. Published research uses various experimental methodologies and metrics that are difficult to compare. It also often fails to sufficiently document the details of proposed algorithms or the evaluations employed. Researchers waste time reimplementing well-known algorithms, and the new implementations may miss key details from the original algorithm or its subsequent refinements. When proposing new algorithms, researchers should compare them against finely-tuned implementations of the leading prior algorithms using state-of-the-art evaluation methodologies. With few exceptions, published algorithmic improvements in our field should be accompanied by working code in a standard framework, including test harnesses to reproduce the described results. To that end, we present the design and freely distributable source code of LensKit, a flexible platform for reproducible recommender systems research. LensKit provides carefully tuned implementations of the leading collaborative filtering algorithms, APIs for common recommender system use cases, and an evaluation framework for performing reproducible offline evaluations of algorithms. We demonstrate the utility of LensKit by replicating and extending a set of prior comparative studies of recommender algorithms — showing limitations in some of the original results — and by investigating a question recently raised by a leader in the recommender systems community on problems with error-based prediction evaluation.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*

General Terms

Algorithms, experimentation, measurement, standardization

Keywords

Recommender systems, implementation, evaluation

RecSys '11, October 23–27, 2011, Chicago, Illinois, USA.
Copyright 2011 ACM 918-1-4503-0683-6/11/10 ...$10.00.

1. INTRODUCTION

It is currently difficult to reproduce and extend recommender systems research results. Algorithmic enhancements are typically published as mathematical formulae rather than working code, and there are often details and edge cases that come up when trying to implement them that are not discussed in the research literature. Further, the state of the art has developed incrementally and in a decentralized fashion, so the original papers for a particular algorithm may not consider important improvements developed later.

As a result, current best practices in recommender systems must be rediscovered and reimplemented for each research project. In the process, important optimizations such as preprocessing normalization steps may be omitted, leading to new algorithms being incorrectly evaluated.

Recommender evaluation is also not handled consistently between publications. Even within the same basic structure, there are many evaluation methods and metrics that have been used. Important details are often omitted or unclear, compounding the difficulty of comparing results between papers or lines of work.

To enable recommender systems research to advance more scientifically and rapidly, we suggest raising the standards of the field to expect algorithmic advances to be accompanied by working, reusable code in a framework that makes re-running evaluations and experiments easy and reliable. This framework (or frameworks) should include world-class implementations of the best previously-known algorithms and support reproducible evaluation with standard datasets using cross-validation and a suite of appropriate metrics.

To foster this movement, we present LensKit, an open source recommender systems toolkit we have developed. LensKit is intended to provide a robust, extensible basis for research and education in recommender systems. It is suitable for deployment in research-scale recommender systems applications, developing and testing new algorithms, experimenting with new evaluation strategies, and classroom use. Its code and surrounding documentation also serves as documentation of what is necessary to take recommender algorithms from the mathematical descriptions in the research literature to actual working code.

In the first portion of this paper, we present the design of LensKit. We then demonstrate it with a comparative evaluation of common recommender algorithms on several data sets and a new experiment examining whether rank-based evaluation methods would improve recommender design. We conclude with a vision for the future of the recommender systems research community and future work for LensKit.

2. OTHER RECOMMENDER TOOLKITS

Over the past two decades, there have been a number of software packages developed for recommendation, including SUGGEST[1], MultiLens, COFI[2], COFE, Apache Mahout[3], MyMediaLite[4], and EasyRec[5], jCOLIBRI[6], and myCBR[7]. Several of these are still under active development.

LensKit sets itself apart by being expressly designed for flexibility and extensibility in research environments, while maintaining high performance; its primary concern is to be maximally useful for research and education, not to support large-scale commercial operations. It is also designed to support a wide variety of recommendation approaches; while our current development focus is on collaborative filtering methods, we have designed the software to support other approaches as well. In addition to algorithms & APIs, LensKit provides a flexible and sophisticated evaluation framework for performing reproducible evaluations of algorithms.

The source code for LensKit is publicly available under the GNU Lesser GPL (version 2 or later), allowing it to be used in a variety of contexts. It is in Java, so it can be used and extended in many environments in a widely-known language.

We encourage the research community to consider implementation on one of the leading recommender platforms as an important step in preparing an algorithm for publication. We propose LensKit as one such platform and believe that it is particularly well-suited for recommender research.

Other research communities have benefited from open platforms providing easy access to the state of the art. Lemur[8] and Lucene[9] 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 [7] 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.

3. DESIGN OF LENSKIT

The design of LensKit is driven by three primary goals:

Modularity. Many recommender algorithms naturally decompose into several constituent pieces, such as normalizers, similarity functions, and prediction rules [8]. Further, many of these components are not specific to one particular algorithm but can be reused in other algorithms. When implementing recommendation methods for LensKit, we design them to be highly modular and reconfigurable. This allows improvements in individual components to be tested easily, and allows the recommender to be completely reconfigured for the particular needs of a target domain.

Clarity. Since LensKit is intended as a platform for recommender systems research and education, we aim for a clear design and well-documented, straightforward (but not naïve) code. Clarity also helps us meet our goals of providing machine-executable documentation of various details in recommender implementations.

LensKit is also usable in real-world situations, particularly web applications, to provide recommender services for live systems and other user-facing research projects. This introduces some complexity in the design, as LensKit must be capable of interacting with data stores and integrating with other application frameworks. Our approach is to keep the core simple but design its interfaces so that the appropriate extension and integration points are available for live systems (e.g. request handlers in a multithreaded web application).

The impact of the integration concern is evident in the interfaces a client uses to create an active recommender: the client first configures and builds a recommender engine, then opens a session connected to the data store, and finally asks for the relevant recommender object. This approach enables recommender models to be computed offline, then recombined with a database connection in a web application request handler. Omitting such considerations would make it difficult to use LensKit recommender implementations outside of batch evaluation settings.

Efficiency. In developing LensKit, we prefer clear code over obscure optimizations, but we seek reasonable efficiency through our choice of data structures and implementation techniques. LensKit is capable of processing large, widely-available data sets such as the MovieLens 10M and the Yahoo! Music data sets on readily available hardware.

LensKit is implemented in Java. Java balances reasonable efficiency with code that is readily readable by a broad audience of computer scientists. Since it runs on the JVM, LensKit is also usable from many other languages such as Groovy, Scala and Ruby (via JRuby).

3.1 Core APIs

The fundamental interfaces LensKit exposes are the `ItemScorer` and `ItemRecommender` interfaces, providing support for the traditional *predict* and *recommend* tasks respectively. Separating these interfaces allows configurations to only support the operations they meaningfully can; a collaborative filter configured and trained on purchase data is likely unsuitable for predicting ratings. `ItemScorer` is a generalization of *predict*, computing general per-user scores for items. A subclass, `RatingPredictor`, scores items by predicted preference in the same scale as user ratings.

Listing 1 shows the core methods exposed by these interfaces. The predict API is fairly self-explanatory. The recommend API allows the recommended items to be controlled via two sets: the *candidate* set C and the *exclude* set E. Only items in $C \backslash E$ are considered for recommendation. Either or both sets may be unspecified; the default candidate set is the set of all recommendable items, while the default exclude set varies by recommender but is generally something in the spirit of "items the user has rated" (this will be different e.g. for recommenders operating on page view and purchase data). This set-based exclusion provides an easy way for client code to integrate recommendation with other data

[1] http://glaros.dtc.umn.edu/gkhome/suggest/overview/

[2] http://savannah.nongnu.org/projects/cofi/

[3] http://mahout.apache.org

[4] http://www.ismll.uni-hildesheim.de/mymedialite/

[5] http://www.easyrec.org/

[6] http://gaia.fdi.ucm.es/projects/jcolibri/

[7] http://mycbr-project.net/

[8] http://www.lemurproject.org

[9] http://lucene.apache.org/

Listing 1: Core LensKit interfaces

```
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.
   */
  ScoredLongList recommend(long user, int count,
      Set<Long> candidates, Set<Long> excludes);
}
```

queries. If the recommender is capable of providing scores with recommendations, the recommendations can be further blended with other data such as search relevance.

3.2 Data Access Layer

LensKit incorporates a simple data access abstraction so that algorithms can access the user history and preference data. Implementations can be easily backed by flat files, databases, persistence architectures such as Hibernate[10], and nontraditional data stores. We provide flat file and in-memory implementations as well as a reconfigurable JDBC-based implementation for database integration. This design is based on the *Data Access Object* J2EE design pattern. Data access objects (DAOs) are session-scoped, corresponding to database connections or persistent sessions; this enables clean integration with web frameworks.

The DAO represents users as having histories of *events*, such as ratings or purchases. Recommenders then extract the event data they can process from the user's history to perform recommendation.

Many aspects of algorithm implementation are easier if we can assume that the LensKit has access to an immutable view of the ratings data while building a recommender model; this allows multiple passes to be taken over it, and recommender components can be developed and trained independently and freely mixed. LensKit therefore takes a snapshot of the data model to be used in the build process. By default, this snapshot is an in-memory copy of the database; DAO implementors can use other methods such as transactions or replication as appropriate.

Iterative methods, such as gradient descent and expectation maximization, require the ability to quickly and repeatedly iterate over the data set. To facilitate this, we provide another abstraction, a *rating snapshot*, to provide an efficient in-memory representation of rating data for rapid iteration in iterative model building methods.

3.3 Building and Using Recommenders

Listing 2 shows the basic flow to construct and use a LensKit recommender implementation. There are a number of steps and classes in use, but this allows LensKit to be a highly flexible in the types of recommenders it supports.

The recommender engine factory manages configuration and builds the recommender engine. Recommender engines encapsulate whatever pre-computed model data an algorithm requires and provide a way to reconnect this model data with the backing data store to produce a working recommender. Algorithm implementations are designed so that data-dependent querying and prediction is separate from pre-computed data. This separation allows recommender engines to be built off-line and shared across database connections.

Listing 2: Building a recommender

```
// Configure data access
daoFactory = new SimpleFileRatingDAO.Factory(fileName);
// Create rec. engine factory
factory = new LenskitRecommenderEngineFactory(daoFactory);
// Configure the recommender
factory.setComponent(ItemRecommender.class,
    ItemItemRecommender.class);
// Build the engine
engine = factory.build();
// Get a recommender
recommender = engine.open();
try {
    // Use recommenders and predictors
    itemRec = recommender.getItemRecommender();
} finally {
    // Close the recommender when finished
    recommender.close();
}
```

3.4 Recommender Configuration

In order to support the degree of modularity we require in LensKit, we needed a flexible configuration system to wire together the various objects needed to provide recommendations. Configuration is particularly complicated in the face of pre-computed models and session-scoped data access (e.g. a user-user recommender that uses a normalization stage dependent on pre-computed values and a similarity function that fetches user demographic information from the database).

Dependency injection [5] provides a powerful and flexible solution to these problems. LensKit's configuration mechanism, implemented in the recommender engine factory, looks at the constructors and setters of the various recommender components to determine their dependencies and figure out how to satisfy them. Java annotations provide guidance to the resolver, specify defaults, and define names for parameters. Annotations also indicate objects that need to be pre-computed as a part of the recommender model.

Client code, therefore, only needs to specify which implementations it needs for particular components. In particular, it needs to specify the `ItemScorer` and/or `ItemRecommender` implementations it wishes to use; most other components have default implementations. The client then specifies any other needed components or parameter values, and the factory automatically constructs all other necessary components.

To manage the configuration, the factory uses several *containers*, environments for resolving dependencies and supplying values and instances. Figure 1 shows these containers. The factory looks at the required predictor and recommender implementations, resolves their dependencies from the configuration and defaults found in annotations, and builds all objects that are marked to be built using the *build container*.

After building all pre-built objects, the factory sets up the *recomender container*, responsible for supplying all components that are shared between sessions. The recommender container contains instances or provision rules for all objects which are not dependent on the DAO. It also constructs the necessary configuration for a complete recommender and

[10] `http://www.hibernate.org`

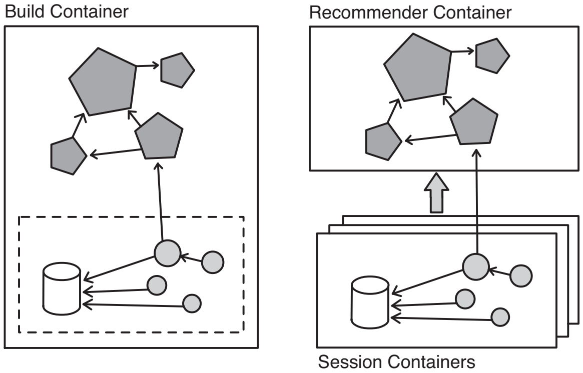


Figure 1: Containers in the recommender lifecycle.

encapsulates this configuration, along with the recommender container, in a recommender engine.

The recommender engine, when a session is opened, creates a new *session container* that inherits configuration and objects from the recommender container and can reconnect those objects with a DAO. The session container is then encapsulated in a `Recommender`, which uses it to provide the final `ItemScorer` and `ItemRecommender` objects.

The result is that building new modular, extensible algorithm implementations is easy. The developer merely needs to implement the core interfaces for the new algorithm and make their objects depend on the appropriate other components. New algorithms will often be able to reuse other components supplied with LensKit. They can define new parameters by creating Java annotations, and LensKit's infrastructure takes care of providing all components with their necessary configuration. It also ensures that components that need data access are instantiated fresh for each session and that expensive objects, such as model data, are automatically shared between all components that need them.

3.5 Implementations and Components

LensKit provides implementations of three commonly-used collaborative filtering algorithms: user-user [15, 8], item-item [16], and regularized gradient descent SVD [6, 14]. These algorithms are each split into components that can be recombined and replaced independently. Section 5 provides more details on configuration points exposed by each algorithm.

The modular design architecture for algorithm implementation accomplishes two important things. First, it enables improvements that only touch one piece of the collaborative filtering process, such as a new normalization step, to be implemented and used in the context of an existing system and tested with multiple algorithms. Second, it allows new algorithms to reuse pieces of existing ones, decreasing the effort needed to implement and test a new algorithm.

LensKit provides several components that can be reused in multiple algorithms. Of particular note are its *baseline predictors* and *normalizers*. Baseline predictors are rating predictors that are guaranteed to be able to generate a prediction for any user-item pair. Supplied baselines include global mean, item mean (falling back to global mean for unknown items), user mean (again falling back to global mean), and item-user mean (item mean plus user's average offset from item mean). Baselines are used by the collaborative filtering algorithms to normalize rating data or to supply predictions when they are unable to [8, 11].

Normalizers apply reversible transformations to a user's rating vector. They are used to normalize data prior to computing similarities or to normalize ratings for prediction generation and denormalize the resulting predictions. Normalization is crucial to good performance with many algorithms [15, 16, 6, 11]. Besides the identity function, we provide baseline-subtracting normalizers (subtracting the baseline predictor from each rating) and a user-variance normalizer that normalizes user ratings to z-scores [8].

4. EVALUATING RECOMMENDERS

In addition to a recommendation API and several collaborative filtering implementations, LensKit provides a framework for offline, data-driven evaluation of recommender algorithms. This allows any algorithm implementing with LensKit to be evaluated and compared against existing approaches.

We currently provide support for train-test and k-fold cross-validation evaluation strategies with a variety of metrics [9, 17] and plan to add support for additional evaluation strategies, such as temporal methods, as development continues. LensKit's common recommender API also provides a good basis for experimenting with new evaluation strategies.

4.1 Importing Data

The evaluation framework accesses rating data stored in an SQLite database. Algorithm definitions can optionally require that the data be pre-loaded into memory if the algorithm does not operate efficiently using database queries.

Most publicly-available data sets are distributed as text files. A delimited text file of user, item, and rating data, possibly with timestamps, is the most common format. LensKit includes an importer to load delimited text files into database tables. This importer has been tested to work on the MovieLens data sets and the music data set from Yahoo! WebScope.

LensKit also provides an importer that splits up rating data for cross-validation. The split is done by partitioning the users into k disjoint sets. For each set, n randomly selected ratings are withheld from each user's profile and emitted as the test set, while the remaining ratings from those users and all other users are stored in the training set. LensKit also provides support for withholding ratings based on timestamp (testing against each user's n most recent ratings).

4.2 Train-Test Evaluations

The train-test prediction evaluator builds a recommender from the train table in a database and measures the recommender's ability to predict the ratings in the test set. Cross-validation can be achieved by running a train-test evaluation multiple times on the output of the crossfold importer.

The evaluator supports multiple pluggable metrics, allowing the recommender to be measured in a variety of ways. LensKit includes the standard accuracy metrics MAE and RMSE (both globally averaged and averaged per-user), as well as coverage metrics and the rank-based metrics nDCG [10] and half-life utility [3].

nDCG is employed as a prediction evaluator by ranking the items in order of prediction and computing the nDCG using the user's rating for each item as its value (half-life utility is implemented similarly). The result is a measurement of the ability of the predictor to rank items consistent with the order imposed by the user's ratings, which should result in accurate top-N recommendations for algorithms that order by prediction. This configuration is a more realistic evaluation than asking the recommender to recommend N items and then computing nDCG, as an algorithm that produces a high-quality recommendation that the user had never seen and rated will not be penalized for doing its job by promoting the unknown item over a known but less-well-liked item.

Combined with statistical analysis scripts in a language like R, this evaluation framework supports fully reproducible evaluations. The ability to download and run the evaluations behind new recommender system research papers holds great promise for increasing the quality and reuseability of research results and presentation.

5. COMPARATIVE ALGORITHM EVALUATION

To demonstrate LensKit we present in this section a comparative evaluation of several design decisions for collaborative filtering algorithms, in the spirit of previous comparisons within a single algorithm [8, 16]. We compare user-user, item-item, and the regularized gradient descent SVD algorithm popularized by Simon Funk [6, 14]. This evaluation extends 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.

5.1 Data and Experimental Setup

We use four data sets for our evaluation: three data sets from MovieLens, containing 100K, 1M and 10M movie ratings (ML-100K, ML-1M and ML-10M), and the Yahoo! Music Ratings data set from Yahoo! WebScope, containing 700M song ratings in 10 segments (Y!M). We use all three MovieLens datasets to emphasize the ability of LensKit to generate results that are directly comparable to previously published work; the results we present here are primarily from ML-1M.

On all three MovieLens data sets, we performed 5-fold cross-validation as described in section 4.1 and averaged the results across the folds. 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.

For each train-test set, we built a recommender algorithm and evaluated its predict performance using MAE, RMSE, and nDCG, as described in section 4.2.

5.2 User-User CF

Our implementation of user-user collaborative filtering provides a number of extension points, allowing the algorithm to be tuned and customized. The similarity function, neighborhood finding strategy, pre-similarity normalization, predictor normalization, and baseline predictor (used when no neighborhood can be computed) can all be independently configured.[11] Each component also exposes various parameters, such as the neighborhood size and smoothing and damping parameters.

Figure 2(a) shows the performance of user-user CF on the ML-1M data set for several similarity functions and neighborhood sizes. We used item-user mean normalization for prediction, resulting in the following prediction rule (μ is the global mean rating, $\mathcal{N}$ the most similar users who have rated i, and $r_{u,i} = \varnothing$ for missing ratings):

$$p_{u,i} = \mu + b_i + b_u + \frac{\sum_{u' \in \mathcal{N}} \mathrm{sim}(u, u') \cdot (r_{u',i} - \mu - b_i - b_{u'})}{\sum_{u' \in \mathcal{N}} |\mathrm{sim}(u, u')|}$$

$$b_i = \frac{\sum_{\{u : r_{u,i} \neq \varnothing\}} (r_{u,i} - \mu)}{|\{u : r_{u,i} \neq \varnothing\}|}$$

$$b_u = \frac{\sum_{\{i : r_{u,i} \neq \varnothing\}} (r_{u,i} - \mu - b_i)}{|\{i : r_{u,i} \neq \varnothing\}|}$$

Previous work suggested using normalizing by the user mean [15] or the user mean and variance [8] to generate predictions, but the item-user mean performed comparably in our experiments on the ML-100K set (slightly underperforming mean-and-variance on MAE but doing slightly better on RMSE). The item-user mean was also used as the baseline for unpredictable items.

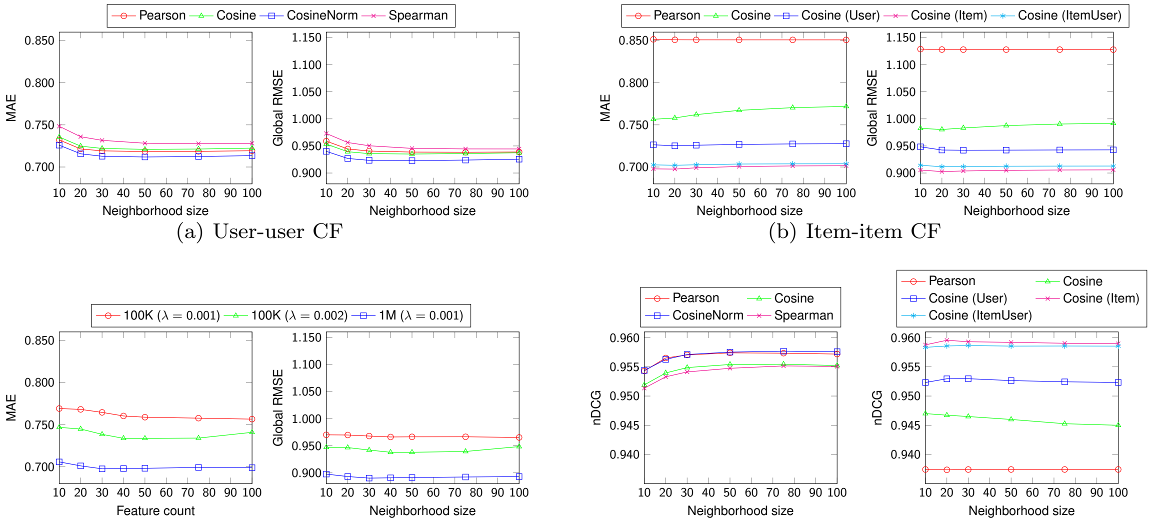
For both Pearson correlation and Spearman rank correlation, we applied a significance weighting threshold of 50 [8], as this improved their performance. Cosine is unadjusted and computed on each user's raw ratings. The CosineNorm similarity is similarly unadjusted but operates on user-mean-centered ratings. Cosine similarity on mean-centered data is identical to Pearson correlation, scaled almost proportionally to $\frac{I_1 \cap I_2}{|I_1||I_2|}$; our data suggest that the self-damping effect of cosine similarity produces better results than applying significance weighting to Pearson correlation. Prior work [3, 8] found cosine to underperform correlation-based similarity functions methods and therefore concluded that it is not a good choice for user-user CF; properly normalized, however, it performs quite well.

5.3 Item-Item CF

Our item-item CF implementation is similarly configurable, although it currently requires the same data normalization to be used both when building the similarity matrix and computing final predictions. This configuration is generally preferable, however, as early trials with differing normalizations produced poor results.

Figure 2(b) shows the performance we achieved with item-item CF. 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 of Sarwar et al. [16]. Consistent with previous work on smaller data sets, normalized cosine performs the best. 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.

[11] There are a few limitations due to using PicoContainer for dependency injection. We are working on a solution to this problem that will allow true independence of all components.



(a) User-user CF

(b) Item-item CF

(c) Regularized (Funk) SVD

(d) User-user CF (nDCG)

(e) Item-item CF (nDCG)

Figure 2: Accuracy of recommender predictions (ML-1M data set unless otherwise indicated)

5.4 Regularized SVD

LensKit provides a regularized gradient descent SVD implementation, also known as FunkSVD. Figure 2(c) shows the performance of this 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 [6], 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 $[1, 5]$ after each feature's contribution is added. This algorithm's implementation is somewhat less flexible than the others, as supporting features such as the rating range clamp requires the normalization phase to be built in to the algorithm implementation. Therefore, it only supports configuring a baseline predictor and not a general normalizer.

5.5 Comparison

Figure 3 shows the relative performance of representative algorithms for each family on the ML-1M, ML-10M, and Y!M data sets. The user-user algorithm uses normalized Cosine similarity and 50 neighbors; it is excluded from Y!M due to its slow performance on large data sets. Item-item uses 30 neighbors and item-user mean normalization. FunkSVD uses 30 latent factors, 100 iterations per factor, range clamping and $\lambda = 0.001$. FunkSVD outperforms the other two algorithms on all three metrics. While we do not have precise timing data, FunkSVD took substantially less time to build its model than item-item on the Y!M data set; on all other data sets, item-item had the fastest build phase.

6. ERROR VS. RANK IN EVALUATION

In a recent blog post [1], Xavier Amatriain raised the question of whether approaches to building and evaluating recommender systems assuming that user ratings are provided on a linear scale (that is, a 5-star movie is as much better than a 4-star as a 4-star is than a 3-star) is fundamentally flawed. His argument, briefly summarized, is this: it is well-known that Likert-style feedback from users is ordinal (users like 5-star movies better than 4-star movies) but not cardinal (it tells us little about *how much* better the 5-star movie is than the 4-star one) [2]. That is, it is usable for ranking items by preference but not for measuring preference in a manner comparable between users. Most recommender algorithms, however, assume that user ratings are a linear scale, though many normalize ratings to compensate for differences in how users use the scale. Further, common evaluations such as MAE or RMSE also assume that rating data is a measurement. Therefore, we may be both measuring and optimizing the wrong thing when we build recommender systems. He suggests that rank-based evaluations such as normalized discounted cumulative gain (nDCG) may measure the ability of recommender algorithms to accurately model user preferences more accurately.

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. To evaluate this question, we tested a selection of recommenders with both nDCG 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.

Figures 2(d) and 2(e) show the various permutations of user-user and item-item collaborative filtering measured using nDCG. There is little difference in the relative performance of the different variants. 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 Amatriain is wrong. 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, will be

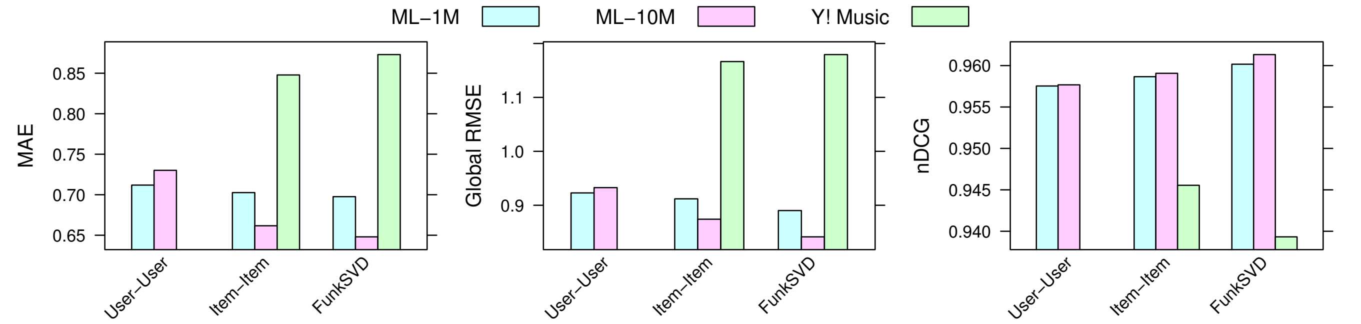


Figure 3: Representative algorithms

needed 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.

7. THE RECOMMENDER ECOSYSTEM

We believe that more extensive publication and documentation of methods is crucial to improving the recommender ecosystem. Publishing the working code for new algorithms, as well as the code to run and evaluate recommenders and synthesize the results, will make it easier to apply and extend research on recommender systems. We see two major aspects of such publication:

- Publishing the source code for algorithmic improvements, ready to run in a common evaluation environment, allows those improvements to be more easily reused and understood. It also serves as a final source for details which may be omitted, either inadvertently or due to space constraints, from the paper.

- Publishing data sets, evaluation methods, and analysis code (e.g. R scripts for producing charts and summaries) allows recommender research to be more readily reproduced and compared.

Open recommender toolkits such as LensKit provide a common, accessible basis for this publication. When authors provide LensKit-compatible implementations of new algorithms, the community can easily try them on other data sets or compare them against new proposed improvements.

To foster this direction in recommender research, we encourage anyone doing new recommender research with LensKit, particularly publishing papers using it, to create a page in the LensKit wiki[12] describing their work providing any relevant code. We are also interested in working with the community to create a general repository for recommender systems research, hosted by us or others, not limited to work done with LensKit but upholding the same standards of public code and reproducible evaluations. As an example, the complete code for reproducing the evaluations in this paper is available[13].

We see this culture not only improving the state of recommender systems research, but also aiding user-based evaluation and other human-recommender interaction research. Making algorithmic enhancements publicly available and easily reusable lowers the bar to using recent developments in deployed systems, user studies, and novel research settings.

There remains the concern of the difficulty of publishing code in restricted environments, such as industry R&D labs. If experiments and algorithms are disclosed in sufficient detail that they can be reimplemented, however, it should not be a problem to complete that disclosure by providing working code. The implementation need not be industrial strength in scale or performance, but should demonstrate a correct implementation of the algorithm as published in the paper[14]. Of course these labs will often have filed a patent application before publishing the paper; though algorithms in the public domain may advance the field more rapidly, we propose no restriction on publishing papers on patented algorithms.

Reproducibility includes datasets in addition to code. What should be done about results that are based on proprietary datasets? We propose that the community sets an expectation that authors of published papers make a version of their dataset available to the research community, so their results can be reproduced. This dataset may be suitably anonymized, require an application to receive the data, and licensed only for non-commercial use, but must be available for legitimate research purposes. In some cases, the data may be so sensitive that not even an anonymized version can be made available. In that case, we propose that authors be expected to demonstrate their results on a suitable public dataset in addition to the proprietary dataset. If the results do not hold up on the public datasets, the reviewers should be encouraged to assume the results are narrow, perhaps only applying to the particular environment in which they were achieved. It may well be that the results are so impressive that even given this limitation the paper should still be published; this decision should be left to the reviewers.

8. CONCLUSION AND FUTURE WORK

It is currently difficult to reproduce and extend research results and algorithmic developments in recommender systems. A culture of open code and reproducible experiments in common frameworks will move the recommender systems

[12] `http://dev.grouplens.org/trac/lenskit/wiki/`

[13] `http://dev.grouplens.org/hg/lenskit-recsys2011`

[14] We should confess that we still receive an average of more than one email a month asking for clarification of some details of a paper we published in 1994 [15], so clearly not all written descriptions of algorithms are sufficient!

community forward by fostering increased reuse and extension and making it easier to compare against prior results. We are putting our money where our mouth is by making available to the community LensKit, the LensKit Wiki, and, as an example of reproducible results in this field, the LensKit scripts to reproduce the research in this paper.

In this paper, we have presented LensKit, an open source recommender systems toolkit intended to foster more open recommender systems research. LensKit is a flexible platform to enable researchers to easily implement and evaluate recommender algorithms. LensKit can also serve as a production recommender for small to medium scale lab or field trials of recommender interfaces, which will benefit from the finely-tuned implementations of the leading algorithms.

We have performed experiments demonstrating that the LensKit implementations meet or beat the canonical implementations of the best-known algorithms. We have further demonstrated that the platform is a valuable tool for addressing open questions in the field.

Development on LensKit is ongoing. In particular, we plan to add several important features in the near future:

- Support for content-based and hybrid recommender systems. The current algorithms in LensKit are all collaborative filtering algorithms, but the API and infrastructure are designed to support more general types of recommenders as well. We will be implementing example hybrid content/collaborative algorithms to demonstrate this capability.
- A framework for learning parameters through cross-validation. Many algorithms have features, such as learning rates and smoothing terms, that may be learned through experimentation. We will provide a general framework to learn parameters, including an evaluation methodology to accurately measure the performance of the resulting automatically-tuned algorithm.
- Additional evaluation strategies. In addition to the train-test prediction evaluation, we will provide temporal evaluation methods, such as time-averaged RMSE [12] and Profile MAE [4], and recommendation list evaluations. We will also extend LensKit with recommender throughput evaluation strategies [13].

Our community is very successful today, but we believe the recommender systems ecosystem approach to research offers an opportunity for even greater success in the future. The research community will benefit from increased publication of working implementations, easier access to the state of the art, and a library of reproducible evaluations on publicly-available data sets. We hope that LensKit and the surrounding community and infrastructure, as well as the other publicly-available recommender toolkits, will allow recommender systems research to be more easily reproduced, extended, and applied in real applications. Please join us!

9. ACKNOWLEDGEMENTS

We gratefully acknowledge our colleagues in GroupLens Research for their support in this project, particularly Jack Kolb and Devyn Coveyou for their coding work and Rich Davies for his invaluable design advice. This work has been funded by NSF grants IIS 05-34939, 08-08692, and 10-17697.

10. REFERENCES

[1] X. Amatriain. Recommender systems: We're doing it (all) wrong. http://technocalifornia.blogspot.com/2011/04/recommender-systems-were-doing-it-all.html, Apr. 2011.

[2] N. W. H. Blaikie. *Analyzing quantitative data: from description to explanation*. SAGE, Mar. 2003.

[3] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *UAI 1998*, pages 43–52. AAAI, 1998.

[4] R. Burke. Evaluating the dynamic properties of recommendation algorithms. In *ACM RecSys '10*, pages 225–228. ACM, 2010.

[5] M. Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, Jan. 2004.

[6] S. Funk. Netflix update: Try this at home. http://sifter.org/~simon/journal/20061211.html, Dec. 2006.

[7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, and P. Reutemann. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[8] J. Herlocker, J. A. Konstan, and J. Riedl. An empirical analysis of design choices in Neighborhood-Based collaborative filtering algorithms. *Inf. Retr.*, 5(4):287–310, 2002.

[9] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, 2004.

[10] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst. (TOIS)*, 20(4):422–446, Oct. 2002.

[11] Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *ACM KDD '08*, pages 426–434. ACM, 2008.

[12] N. Lathia, S. Hailes, and L. Capra. Evaluating collaborative filtering over time. In *SIGIR '09 Workshop on the Future of IR Evaluation*, July 2009.

[13] J. Levandoski, M. Ekstrand, J. Riedl, and M. Mokbel. RecBench: benchmarks for evaluating performance of recommender system architectures. In *VLDB 2011*, 2011.

[14] A. Paterek. Improving regularized singular value decomposition for collaborative filtering. In *KDD Cup and Workshop 2007*, Aug. 2007.

[15] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: an open architecture for collaborative filtering of netnews. In *ACM CSCW '94*, pages 175–186. ACM, 1994.

[16] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In *ACM WWW '01*, pages 285–295. ACM, 2001.

[17] G. Shani and A. Gunawardana. Evaluating recommendation systems. In F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, editors, *Recommender Systems Handbook*, pages 257–297. Springer, 2010.