recommenderlab: A Framework for Developing and Testing Recommendation Algorithms

Michael Hahsler
SMU

Abstract

The problem of creating recommendations given a large data base from directly elicited ratings (e.g., ratings of 1 through 5 stars) is a popular research area which was lately boosted by the Netflix Prize competition. While several libraries which implement recommender algorithms have been developed over the last decade there is still the need for a framework which facilitates research on recommender systems by providing a common development and evaluation environment. This paper describes recommenderlab which provides the infrastructure to develop and test recommender algorithms for rating data and 0-1 data in a unified framework. The Package provides basic algorithms and allows the user to develop and use his/her own algorithms in the framework via a simple registration procedure.

Keywords: recommender systems, collaborative filtering, evaluation.

1. Introduction

Predicting ratings and creating personalized recommendations for products like books, songs or movies online came a long way from Information Lense, the first system using social filtering created by Malone, Grant, Turbak, Brobst, and Cohen (1987) more than 20 years ago. Today recommender systems are an accepted technology used by market leaders in several industries (e.g., by Amazon\(^1\), Netflix\(^2\) and Pandora\(^3\)). Recommender systems apply statistical and knowledge discovery techniques to the problem of making product recommendations based on previously recorded data (Sarwar, Karypis, Konstan, and Riedl 2000). Such recommendations can help to improve the conversion rate by helping the customer to find products she/he wants to buy faster, promote cross-selling by suggesting additional products and can improve customer loyalty through creating a value-added relationship (Schafer, Konstan, and Riedl 2001). The importance and the economic impact of research in this field is reflected by the Netflix Prize\(^4\), a challenge to improve the predictions of Netflix’s movie recommender system by more than 10% in terms of the root mean square error. The grand price of 1 million dollar was awarded in September 2009 to the Belcore Pragmatic Chaos team.  

Ansari, Essegaier, and Kohli (2000) categorizes recommender systems into content-based approaches and collaborative filtering. Content-based approaches are based on the idea that if we

\(^1\)http://www.amazon.com  
\(^2\)http://www.netflix.com  
\(^3\)http://www.pandora.com  
\(^4\)http://www.netflixprize.com/
can elicit the preference structure of a customer (user) concerning product (item) attributes then we can recommend items which rank high for the user’s most desirable attributes. Typically, the preference structure can be elicited by analyzing which items the user prefers. For example, for movies the Internet Movie Database\(^5\) contains a wide range of attributes to describe movies including genre, director, write, cast, storyline, etc. For music, Pandora, a personalized online radio station, creates a stream of music via content-based recommendations based on a system of hundreds of attributes to describe the essence of music at the fundamental level including rhythm, feel, influences, instruments and many more (John 2006).

In recommenderlab we concentrate on the second category of recommender algorithms called collaborative filtering. The idea is that given rating data by many users for many items (e.g., 1 to 5 stars for movies elicited directly from the users), one can predict a user’s rating for an item not known to her or him (see, e.g., Goldberg, Nichols, Oki, and Terry 1992) or create for a user a so called top-$N$ lists of recommended items (see, e.g., Deshpande and Karypis 2004). The premise is that users who agreed on the rating for some items typically also agree on the rating for other items.

Several projects were initiated to implement recommender algorithms for collaborative filtering. Table 1 gives an overview of open-source projects which provide code which can be used by researchers. The extend of (currently available) functionality as well as the target usage of the software packages vary greatly. Crab, easyrec, MyMediaLite and Vogoo PHP LIB aim at providing simple recommender systems to be easily integrated into web sites. SVDFeature focuses only on matrix factorization. Cofi provides a Java package which implements many collaborative filtering algorithms (active development ended 2005). LensKit is a relatively new software package with the aim to provide reference implementations for common collaborative filtering algorithms. This software has not reached a stable version at the time this paper was written (October, 2011). Finally, Apache Mahout, a machine learning library aimed to be scalable to large data sets incorporated collaborative filtering algorithms formerly developed under the name Taste.

The R extension package recommenderlab described in this paper has a completely different goal to the existing software packages. It is not a library to create recommender applications but provides a general research infrastructure for recommender systems. The focus is on consistent and efficient data handling, easy incorporation of algorithms (either implemented in R or interfacing existing algorithms), experiment set up and evaluation of the results.

This paper is structured as follows. Section 2 introduces collaborative filtering and some of its popular algorithms. In section 3 we discuss the evaluation of recommender algorithms. We introduce the infrastructure provided by recommenderlab in section 4. In section 5 we illustrate the capabilities on the package to create and evaluate recommender algorithms. We conclude with section 6.

To cite package ‘recommenderlab’ in publications use:

https://github.com/mhahsler/recommenderlab

\(^5\)http://www.imdb.com/
<table>
<thead>
<tr>
<th>Software</th>
<th>Description</th>
<th>Language</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cofi</td>
<td>Collaborative filtering library</td>
<td>Java</td>
<td><a href="http://www.nongnu.org/cofi/">http://www.nongnu.org/cofi/</a></td>
</tr>
<tr>
<td>Crab</td>
<td>Components to create recommender systems</td>
<td>Python</td>
<td><a href="https://github.com/muricoca/crab">https://github.com/muricoca/crab</a></td>
</tr>
<tr>
<td>PREA</td>
<td>Personalized Recommendation Algorithms Toolkit</td>
<td>Java</td>
<td><a href="http://prea.gatech.edu/">http://prea.gatech.edu/</a></td>
</tr>
</tbody>
</table>

Table 1: Recommender System Software freely available for research.
2. Collaborative Filtering

Collaborative filtering (CF) uses given rating data by many users for many items as the basis for predicting missing ratings and/or for creating a top-N recommendation list for a given user, called the active user. Formally, we have a set of users \( U = \{ u_1, u_2, \ldots, u_m \} \) and a set of items \( I = \{ i_1, i_2, \ldots, i_n \} \). Ratings are stored in a \( m \times n \) user-item rating matrix \( R = (r_{jk}) \) where each row represents a user \( u_j \) with \( 1 \leq j \leq m \) and columns represent items \( i_k \) with \( 1 \leq k \leq n \). \( r_{jk} \) represents the rating of user \( u_j \) for item \( i_k \). Typically only a small fraction of ratings are known and for many cells in \( R \) the values are missing. Many algorithms operate on ratings on a specific scale (e.g., 1 to 5 (stars)) and estimated ratings are allowed to be within an interval of matching range (e.g., \([1, 5]\)). From this point of view recommender systems solve a regression problem.

The aim of collaborative filtering is to create recommendations for a user called the active user \( u_a \in U \). We define the set of items unknown to user \( u_a \) as \( I_a = I \setminus \{ i_l \in I | r_{al} = 1 \} \). The two typical tasks are to predict ratings for all items in \( I_a \) or to create a list of the best \( N \) recommendations (i.e., a top-N recommendation list) for \( u_a \). Formally, predicting all missing ratings is calculating a complete row of the rating matrix \( \hat{r}_a \cdot \) where the missing values for items in \( I_a \) are replaced by ratings estimated from other data in \( R \). The estimated ratings are in the same range as the original rating (e.g., in the range \([1, 5]\) for a five star rating scheme).

Creating a top-N list (Sarwar, Karypis, Konstan, and Riedl 2001) can be seen as a second step after predicting ratings for all unknown items in \( I_a \) and then taking the \( N \) items with the highest predicted ratings. A list of top-N recommendations for a user \( u_a \) is an partially ordered set \( T_N = (X, \geq) \), where \( X \subset I_a \) and \( |X| \leq N \) (\(|\cdot|\) denotes the cardinality of the set). Note that there may exist cases where top-N lists contain less than \( N \) items. This can happen if \( |I_a| < N \) or if the CF algorithm is unable to identify \( N \) items to recommend. The binary relation \( \geq \) is defined as \( x \geq y \) if and only if \( \hat{r}_{ax} \geq \hat{r}_{ay} \) for all \( x, y \in X \). Furthermore we require that \( \forall x \in X \forall y \in I_a \hat{r}_{ax} \geq \hat{r}_{ay} \) to ensure that the top-N list contains only the items with the highest estimated rating.

Typically we deal with a very large number of items with unknown ratings which makes first predicting rating values for all of them computationally expensive. Some approaches (e.g., rule based approaches) can predict the top-N list directly without considering all unknown items first.

Collaborative filtering algorithms are typically divided into two groups, memory-based CF and model-based CF algorithms (Breese, Heckerman, and Kadie 1998). Memory-based CF use the whole (or at least a large sample of the) user database to create recommendations.
The most prominent algorithm is user-based collaborative filtering. The disadvantages of this approach is scalability since the whole user database has to be processed online for creating recommendations. Model-based algorithms use the user database to learn a more compact model (e.g., clusters with users of similar preferences) that is later used to create recommendations.

In the following we will present the basics of well known memory and model-based collaborative filtering algorithms. Further information about these algorithms can be found in the recent survey book chapter by Desrosiers and Karypis (2011).

2.1. User-based Collaborative Filtering

User-based CF (Goldberg et al. 1992; Resnick, Iacovou, Suchak, Bergstrom, and Riedl 1994; Shardanand and Maes 1995) is a memory-based algorithm which tries to mimics word-of-mouth by analyzing rating data from many individuals. The assumption is that users with similar preferences will rate items similarly. Thus missing ratings for a user can be predicted by first finding a neighborhood of similar users and then aggregate the ratings of these users to form a prediction.

The neighborhood is defined in terms of similarity between users, either by taking a given number of most similar users (k nearest neighbors) or all users within a given similarity threshold. Popular similarity measures for CF are the Pearson correlation coefficient and the Cosine similarity. These similarity measures are defined between two users $u_x$ and $u_y$ as

\[
\text{sim}_{\text{Pearson}}(\vec{x}, \vec{y}) = \frac{\sum_{i \in I}(\vec{x}_i \bar{x})(\vec{y}_i \bar{y})}{(|I| - 1) \text{sd}(\vec{x}) \text{sd}(\vec{y})} \tag{1}
\]

and

\[
\text{sim}_{\text{Cosine}}(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| \|\vec{y}\|}, \tag{2}
\]

where $\vec{x} = r_x$ and $\vec{y} = r_y$ represent the row vectors in $R$ with the two users’ profile vectors. $\text{sd}(\cdot)$ is the standard deviation and $\|\cdot\|$ is the $l^2$-norm of a vector. For calculating similarity using rating data only the dimensions (items) are used which were rated by both users.

Now the neighborhood for the active user $N(a) \subset U$ can be selected by either a threshold on the similarity or by taking the $k$ nearest neighbors. Once the users in the neighborhood are found, their ratings are aggregated to form the predicted rating for the active user. The easiest form is to just average the ratings in the neighborhood.

\[
\hat{r}_{aj} = \frac{1}{|N(a)|} \sum_{i \in N(a)} r_{ij} \tag{3}
\]

An example of the process of creating recommendations by user-based CF is shown in Figure 1. To the left is the rating matrix $R$ with 6 users and 8 items and ratings in the range 1 to 5 (stars). We want to create recommendations for the active user $u_a$ shown at the bottom of the matrix. To find the $k$-neighborhood (i.e., the $k$ nearest neighbors) we calculate the similarity between the active user and all other users based on their ratings in the database and then select the $k$ users with the highest similarity. To the right in Figure 1 we see a 2-dimensional representation of the similarities (users with higher similarity are displayed closer) with the active user in the center. The $k = 3$ nearest neighbors ($u_4, u_1$ and $u_3$) are selected and
marked in the database to the left. To generate an aggregated estimated rating, we compute the average ratings in the neighborhood for each item not rated by the active user. To create a top-$N$ recommendation list, the items are ordered by predicted rating. In the small example in Figure 1 the order in the top-$N$ list (with $N \geq 4$) is $i_2, i_1, i_7$ and $i_5$. However, for a real application we probably would not recommend items $i_7$ and $i_5$ because of their low ratings.

The fact that some users in the neighborhood are more similar to the active user than others can be incorporated as weights into Equation (3).

$$\hat{r}_{aj} = \frac{1}{\sum_{i \in N(a)} s_{ai}} \sum_{i \in N(a)} s_{ai} r_{ij}$$

$s_{ai}$ is the similarity between the active user $u_a$ and user $u_i$ in the neighborhood.

For some types of data the performance of the recommender algorithm can be improved by removing user rating bias. This can be done by normalizing the rating data before applying the recommender algorithm. Any normalization function $h : \mathbb{R}^{n \times m} \mapsto \mathbb{R}^{n \times m}$ can be used for preprocessing. Ideally, this function is reversible to map the predicted rating on the normalized scale back to the original rating scale. Normalization is used to remove individual rating bias by users who consistently always use lower or higher ratings than other users. A popular method is to center the rows of the user-item rating matrix by

$$h(r_{ui}) = r_{ui} - \bar{r}_u,$$

where $\bar{r}_u$ is the mean of all available ratings in row $u$ of the user-item rating matrix $R$.

Other methods like Z-score normalization which also takes rating variance into account can be found in the literature (see, e.g., Desrosiers and Karypis 2011).

The two main problems of user-based CF are that the whole user database has to be kept in memory and that expensive similarity computation between the active user and all other users in the database has to be performed.
2.2. Item-based Collaborative Filtering

Item-based CF (Kitts, Freed, and Vrieze 2000; Sarwar et al. 2001; Linden, Smith, and York 2003; Deshpande and Karypis 2004) is a model-based approach which produces recommendations based on the relationship between items inferred from the rating matrix. The assumption behind this approach is that users will prefer items that are similar to other items they like.

The model-building step consists of calculating a similarity matrix containing all item-to-item similarities using a given similarity measure. Popular are again Pearson correlation and Cosine similarity. All pairwise similarities are stored in a \( n \times n \) similarity matrix \( S \). To reduce the model size to \( n \times k \) with \( k \ll n \), for each item only a list of the \( k \) most similar items and their similarity values are stored. The \( k \) items which are most similar to item \( i \) is denoted by the set \( S(i) \) which can be seen as the neighborhood of size \( k \) of the item. Retaining only \( k \) similarities per item improves the space and time complexity significantly but potentially sacrifices some recommendation quality (Sarwar et al. 2001).

To make a recommendation based on the model we use the similarities to calculate a weighted sum of the user’s ratings for related items.

\[
\hat{r}_{ai} = \frac{1}{\sum_{j \in S(i) \cap \{l; r_{ai} \neq ?\}} s_{ij}} \sum_{j \in S(i) \cap \{l; r_{ai} \neq ?\}} s_{ij} r_{aj}
\]

Figure 2 shows an example for \( n = 8 \) items with \( k = 3 \). For the similarity matrix \( S \) only the \( k = 3 \) largest entries are stored per row (these entries are marked using bold face). For the example we assume that we have ratings for the active user for items \( i_1, i_5 \) and \( i_8 \). The rows corresponding to these items are highlighted in the item similarity matrix. We can now compute the weighted sum using the similarities (only the reduced matrix with the \( k = 3 \) highest ratings is used) and the user’s ratings. The result (below the matrix) shows that \( i_3 \) has the highest estimated rating for the active user.

Similar to user-based recommender algorithms, user-bias can be reduced by first normalizing the user-item rating matrix before computing the item-to-item similarity matrix.

Item-based CF is more efficient than user-based CF since the model (reduced similarity matrix) is relatively small \( (N \times k) \) and can be fully precomputed. Item-based CF is known to
only produce slightly inferior results compared to user-based CF and higher order models which take the joint distribution of sets of items into account are possible (Deshpande and Karypis 2004). Furthermore, item-based CF is successfully applied in large scale recommender systems (e.g., by Amazon.com).

2.3. User and Item-Based CF using 0-1 Data

Less research is available for situations where no large amount of detailed directly elicited rating data is available. However, this is a common situation and occurs when users do not want to directly reveal their preferences by rating an item (e.g., because it is too time consuming). In this case preferences can only be inferred by analyzing usage behavior. For example, we can easily record in a supermarket setting what items a customer purchases. However, we do not know why other products were not purchased. The reason might be one of the following.

- The customer does not need the product right now.
- The customer does not know about the product. Such a product is a good candidate for recommendation.
- The customer does not like the product. Such a product should obviously not be recommended.

Mild and Reutterer (2003) and Lee, Jun, Lee, and Kim (2005) present and evaluate recommender algorithms for this setting. The same reasoning is true for recommending pages of a website given click-stream data. Here we only have information about which pages were viewed but not why some pages were not viewed. This situation leads to binary data or more exactly to 0-1 data where 1 means that we inferred that the user has a preference for an item and 0 means that either the user does not like the item or does not know about it. Pan, Zhou, Cao, Liu, Lukose, Scholz, and Yang (2008) call this type of data in the context of collaborative filtering analogous to similar situations for classifiers one-class data since only the 1-class is pure and contains only positive examples. The 0-class is a mixture of positive and negative examples.

In the 0-1 case with \( r_{jk} \in \{0, 1\} \) where we define:

\[
\begin{equation}
    r_{jk} = \begin{cases} 
    1 & \text{user } u_j \text{ is known to have a preference for item } i_k \\
    0 & \text{otherwise.}
\end{cases}
\end{equation}
\]

Two strategies to deal with one-class data is to assume all missing ratings (zeros) are negative examples or to assume that all missing ratings are unknown. In addition, Pan et al. (2008) propose strategies which represent a trade-off between the two extreme strategies based on weighted low rank approximations of the rating matrix and on negative example sampling which might improve results across all recommender algorithms.

If we assume that users typically favor only a small fraction of the items and thus most items with no rating will be indeed negative examples, then we have no missing values and can use the approaches described above for real valued rating data. However, if we assume all zeroes are missing values, then this lead to the problem that we cannot compute similarities
using Pearson correlation or Cosine similarity since the not missing parts of the vectors only
contains ones. A similarity measure which only focuses on matching ones and thus prevents
the problem with zeroes is the Jaccard index:

\[ \text{sim}_{\text{Jaccard}}(\mathcal{X}, \mathcal{Y}) = \frac{\left| \mathcal{X} \cap \mathcal{Y} \right|}{\left| \mathcal{X} \cup \mathcal{Y} \right|}, \]

where \( \mathcal{X} \) and \( \mathcal{Y} \) are the sets of the items with a 1 in user profiles \( u_a \) and \( u_b \), respectively. The Jaccard index can be used between users for user-based filtering and between items for
item-based filtering as described above.

### 2.4. Recommendations for 0-1 Data Based on Association Rules

Recommender systems using association rules produce recommendations based on a depen-
dency model for items given by a set of association rules (Fu, Budzik, and Hammond 2000;
Mobasher, Dai, Luo, and Nakagawa 2001; Geyer-Schulz, Hahsler, and Jahn 2002; Lin, Alvarez,
and Ruiz 2002; Demiriz 2004). The binary profile matrix \( \mathbf{R} \) is seen as a database where each
user is treated as a transaction that contains the subset of items in \( \mathcal{I} \) with a rating of 1. Hence
transaction \( k \) is defined as \( T_k = \{ i_j \in \mathcal{I} | r_{jk} = 1 \} \) and the whole transaction data base
is \( \mathcal{D} = \{ T_1, T_2, \ldots, T_U \} \) where \( U \) is the number of users. To build the dependency model, a set
of association rules \( \mathcal{R} \) is mined from \( \mathbf{R} \). Association rules are rules of the form \( \mathcal{X} \rightarrow \mathcal{Y} \) where
\( \mathcal{X}, \mathcal{Y} \subseteq \mathcal{I} \) and \( \mathcal{X} \cap \mathcal{Y} = \emptyset \). For the model we only use association rules with a single item in
the right-hand-side of the rule (\(| \mathcal{Y} | = 1 \)). To select a set of useful association rules, thresholds
on measures of significance and interestingness are used. Two widely applied measures are:

\[
\begin{align*}
\text{support}(\mathcal{X} \rightarrow \mathcal{Y}) &= \text{support}(\mathcal{X} \cup \mathcal{Y}) = \frac{\text{Freq}(\mathcal{X} \cup \mathcal{Y})}{|\mathcal{D}|} = \hat{P}(E_{\mathcal{X}} \cap E_{\mathcal{Y}}) \quad \text{and} \\
\text{confidence}(\mathcal{X} \rightarrow \mathcal{Y}) &= \frac{\text{support}(\mathcal{X} \cup \mathcal{Y})}{\text{support}(\mathcal{X})} = \frac{\hat{P}(E_{\mathcal{Y}} | E_{\mathcal{X}})}{\hat{P}(E_{\mathcal{X}})} \quad \text{(7)}
\end{align*}
\]

Freq(\( \mathcal{I} \)) gives the number of transactions in the data base \( \mathcal{D} \) that contains all items in \( \mathcal{I} \). \( E_{\mathcal{I}} \)
is the event that the itemset \( \mathcal{I} \) is contained in a transaction.

We now require \( \text{support}(\mathcal{X} \rightarrow \mathcal{Y}) > s \) and \( \text{confidence}(\mathcal{X} \rightarrow \mathcal{Y}) > c \) and also include a length
constraint \(| \mathcal{X} \cup \mathcal{Y} | \leq l \). The set of rules \( \mathcal{R} \) that satisfy these constraints form the dependency
model. Although finding all association rules given thresholds on support and confidence is
a hard problem (the model grows in the worse case exponential with the number of items),
algorithms that efficiently find all rules in most cases are available (e.g., Agrawal and Srikant
1994; Zaki 2000; Han, Pei, Yin, and Mao 2004). Also model size can be controlled by \( l, s \) and
\( c \).

To make a recommendation for an active user \( u_a \) given the set of items \( \mathcal{T}_a \) the user likes and
the set of association rules \( \mathcal{R} \) (dependency model), the following steps are necessary:

1. Find all matching rules \( \mathcal{X} \rightarrow \mathcal{Y} \) for which \( \mathcal{X} \subseteq \mathcal{T}_a \) in \( \mathcal{R} \).
2. Recommend \( N \) unique right-hand-sides \( \mathcal{Y} \) of the matching rules with the highest con-
fidence (or another measure of interestingness).

The dependency model is very similar to item-based CF with conditional probability-based
similarity (Deshpande and Karypis 2004). It can be fully precomputed and rules with more
than one items in the left-hand-side ($X$), it incorporates higher order effects between more than two items.

2.5. Other collaborative filtering methods

Over time several other model-based approaches have been developed. A popular simple item-based approach is the *Slope One* algorithm (Lemire and Maclachlan 2005). Another family of algorithms is based on latent factors approach using matrix decomposition (Koren, Bell, and Volinsky 2009). These algorithms are outside the scope of this introductory paper.

3. Evaluation of Recommender Algorithms

Evaluation of recommender systems is an important topic and reviews were presented by Herlocker, Konstan, Terveen, and Riedl (2004) and Gunawardana and Shani (2009). Typically, given a rating matrix $R$, recommender algorithms are evaluated by first partitioning the users (rows) in $R$ into two sets $U_{\text{train}} \cup U_{\text{test}} = U$. The rows of $R$ corresponding to the training users $U_{\text{train}}$ are used to learn the recommender model. Then each user $u_a \in U_{\text{test}}$ is seen as an active user, however, before creating recommendations some items are withheld from the profile $r_{ua}$, and it measured either how well the predicted rating matches the withheld value or, for top-$N$ algorithms, if the items in the recommended list are rated highly by the user.

It is assumed that if a recommender algorithm performed better in predicting the withheld items, it will also perform better in finding good recommendations for unknown items.

To determine how to split $U$ into $U_{\text{train}}$ and $U_{\text{test}}$ we can use several approaches (Kohavi 1995).

- **Splitting**: We can randomly assign a predefined proportion of the users to the training set and all others to the test set.

- **Bootstrap sampling**: We can sample from $U_{\text{test}}$ with replacement to create the training set and then use the users not in the training set as the test set. This procedure has the advantage that for smaller data sets we can create larger training sets and still have users left for testing.

- **k-fold cross-validation**: Here we split $U$ into $k$ sets (called folds) of approximately the same size. Then we evaluate $k$ times, always using one fold for testing and all other folds for leaning. The $k$ results can be averaged. This approach makes sure that each user is at least once in the test set and the averaging produces more robust results and error estimates.

The items withheld in the test data are randomly chosen. Breese et al. (1998) introduced the four experimental protocols called *Given 2*, *Given 5*, *Given 10* and *All-but-1*. For the *Given* $x$ protocols for each user $x$ randomly chosen items are given to the recommender algorithm and the remaining items are withheld for evaluation. For All but $x$ the algorithm gets all but $x$ withheld items.

In the following we discuss the evaluation of predicted ratings and then of top-$N$ recommendation lists.
3.1. Evaluation of predicted ratings

A typical way to evaluate a prediction is to compute the deviation of the prediction from the true value. This is the basis for the Mean Average Error (MAE)

$$\text{MAE} = \frac{1}{|K|} \sum_{(i,j) \in K} |r_{ij} - \hat{r}_{ij}|,$$

(8)

where $K$ is the set of all user-item pairings $(i, j)$ for which we have a predicted rating $\hat{r}_{ij}$ and a known rating $r_{ij}$ which was not used to learn the recommendation model.

Another popular measure is the Root Mean Square Error (RMSE).

$$\text{RMSE} = \sqrt{\frac{\sum_{(i,j) \in K} (r_{ij} - \hat{r}_{ij})^2}{|K|}}$$

(9)

RMSE penalizes larger errors stronger than MAE and thus is suitable for situations where small prediction errors are not very important.

3.2. Evaluation Top-$N$ recommendations

The items in the predicted top-$N$ lists and the withheld items liked by the user (typically determined by a simple threshold on the actual rating) for all test users $U_{test}$ can be aggregated into a so called confusion matrix depicted in table 2 (see Kohavi and Provost (1998)) which corresponds exactly to the outcomes of a classical statistical experiment. The confusion matrix shows how many of the items recommended in the top-$N$ lists (column predicted positive; $d + b$) were withheld items and thus correct recommendations (cell $d$) and how many where potentially incorrect (cell $b$). The matrix also shows how many of the not recommended items (column predicted negative; $a + c$) should have actually been recommended since they represent withheld items (cell $c$).

From the confusion matrix several performance measures can be derived. For the data mining task of a recommender system the performance of an algorithm depends on its ability to learn significant patterns in the data set. Performance measures used to evaluate these algorithms have their root in machine learning. A commonly used measure is accuracy, the fraction of correct recommendations to total possible recommendations.

$$\text{Accuracy} = \frac{\text{correct recommendations}}{\text{total possible recommendations}} = \frac{a + d}{a + b + c + d}$$

(10)

A common error measure is the mean absolute error (MAE, also called mean absolute deviation.
Developing and Testing Recommendation Algorithms

or MAD.

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |\epsilon_i| = \frac{b + c}{a + b + c + d},$$

(11)

where $N = a + b + c + d$ is the total number of items which can be recommended and $|\epsilon_i|$ is the absolute error of each item. Since we deal with 0-1 data, $|\epsilon_i|$ can only be zero (in cells $a$ and $d$ in the confusion matrix) or one (in cells $b$ and $c$). For evaluation recommender algorithms for rating data, the root mean square error is often used. For 0-1 data it reduces to the square root of MAE.

Recommender systems help to find items of interest from the set of all available items. This can be seen as a retrieval task known from information retrieval. Therefore, standard information retrieval performance measures are frequently used to evaluate recommender performance. **Precision** and **recall** are the best known measures used in information retrieval (Salton and McGill 1983; van Rijsbergen 1979).

$$\text{Precision} = \frac{\text{correctly recommended items}}{\text{total recommended items}} = \frac{d}{b + d}$$

(12)

$$\text{Recall} = \frac{\text{correctly recommended items}}{\text{total useful recommendations}} = \frac{d}{c + d}$$

(13)

Often the number of total useful recommendations needed for recall is unknown since the whole collection would have to be inspected. However, instead of the actual total useful recommendations often the total number of known useful recommendations is used. Precision and recall are conflicting properties, high precision means low recall and vice versa. To find an optimal trade-off between precision and recall a single-valued measure like the E-measure (van Rijsbergen 1979) can be used. The parameter $\alpha$ controls the trade-off between precision and recall.

$$\text{E-measure} = \frac{1}{\alpha(1/\text{Precision}) + (1 - \alpha)(1/\text{Recall})}$$

(14)

A popular single-valued measure is the F-measure. It is defined as the harmonic mean of precision and recall.

$$\text{F-measure} = \frac{2 \text{Precision} \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2}{1/\text{Precision} + 1/\text{Recall}}$$

(15)

It is a special case of the E-measure with $\alpha = .5$ which places the same weight on both, precision and recall. In the recommender evaluation literature the F-measure is often referred to as the measure $F1$.

Another method used in the literature to compare two classifiers at different parameter settings is the Receiver Operating Characteristic (ROC). The method was developed for signal detection and goes back to the Swets model (van Rijsbergen 1979). The ROC-curve is a plot of the system’s probability of detection (also called sensitivity or true positive rate TPR which is equivalent to recall as defined in formula 13) by the probability of false alarm (also called false positive rate FPR or $1 - \text{specificity}$, where specificity $= \frac{a}{a + b}$) with regard to model parameters. A possible way to compare the efficiency of two systems is by comparing the size of the area under the ROC-curve, where a bigger area indicates better performance.
4. Recommenderlab Infrastructure

`recommenderlab` is implemented using formal classes in the S4 class system. Figure 3 shows the main classes and their relationships.

The package uses the abstract `ratingMatrix` to provide a common interface for rating data. `ratingMatrix` implements many methods typically available for matrix-like objects. For example, `dim()`, `dimnames()`, `colCounts()`, `rowCounts()`, `colMeans()`, `rowMeans()`, `colSums()` and `rowSums()`. Additionally `sample()` can be used to sample from users (rows) and `image()` produces an image plot.

For `ratingMatrix` we provide two concrete implementations `realRatingMatrix` and `binaryRatingMatrix` to represent different types of rating matrices R. `realRatingMatrix` implements a rating matrix with real valued ratings stored in sparse format defined in package `Matrix`. Sparse matrices in `Matrix` typically do not store 0s explicitly, however for `realRatingMatrix` we use these sparse matrices such that instead of 0s, NAs are not explicitly stored.

`binaryRatingMatrix` implements a 0-1 rating matrix using the implementation of `itemMatrix` defined in package `arules`. `itemMatrix` stores only the ones and internally uses a sparse representation from package `Matrix`. With this class structure `recommenderlab` can be easily extended to other forms of rating matrices with different concepts for efficient storage in the future.

Class `Recommender` implements the data structure to store recommendation models. The creator method

```
Recommender(data, method, parameter = NULL)
```

takes data as a `ratingMatrix`, a method name and some optional parameters for the method and returns a `Recommender` object. Once we have a recommender object, we can predict top-N recommendations for active users using

```
predict(object, newdata, n=10, type=c("topNList", "ratings", "ratingMatrix"), ...).
```
Predict can return either top-\(N\) lists (default setting) or predicted ratings. \texttt{object} is the recommender object, \texttt{newdata} is the data for the active users. For top-\(N\) lists \(n\) is the maximal number of recommended items in each list and \texttt{predict()} will return an objects of class \texttt{topNList} which contains one top-\(N\) list for each active user. For "\texttt{ratings}" and "\texttt{ratingMatrix}", \(n\) is ignored and an object of \texttt{realRatingMatrix} is returned. Each row contains the predicted ratings for one active user. The difference is, that for "\texttt{ratings}" the items for which a rating exists in \texttt{newdata} have a \texttt{NA} instead of a predicted/actual ratings.

The actual implementations for the recommendation algorithms are managed using the registry mechanism provided by package \texttt{registry}. The registry called \texttt{recommenderRegistry} and stores recommendation method names and a short description. Generally, the registry mechanism is hidden from the user and the creator function \texttt{Recommender()} uses it in the background to map a recommender method name to its implementation. However, the registry can be directly queried by

\[
\texttt{recommenderRegistry$get\_entries()}
\]

and new recommender algorithms can be added by the user. We will give and example for this feature in the examples section of this paper.

To evaluate recommender algorithms package \texttt{recommenderlab} provides the infrastructure to create and maintain evaluation schemes stored as an object of class \texttt{evaluationScheme} from rating data. The creator function

\[
\texttt{evaluationScheme(data, method="split", train=0.9, k=10, given=3)}
\]

creates the evaluation scheme from a data set using a method (e.g., simple split, bootstrap sampling, \(k\)-fold cross validation). Testing is performed by withholding items (parameter \texttt{given}). Breese \textit{et al.} (1998) introduced the four experimental withholding protocols called \texttt{Given 2}, \texttt{Given 5}, \texttt{Given 10} and \texttt{All-but-1}. During testing, the \texttt{Given} \(x\) protocol presents the algorithm with only \(x\) randomly chosen items for the test user, and the algorithm is evaluated by how well it is able to predict the withheld items. For \texttt{All-but-}\(x\), a generalization of \texttt{All-but-1}, the algorithm sees all but \(x\) withheld ratings for the test user. \texttt{given} controls \(x\) in the evaluations scheme. Positive integers result in a \texttt{Given} \(x\) protocol, while negative values produce a \texttt{All-but-}\(x\) protocol.

The function \texttt{evaluate()} is then used to evaluate several recommender algorithms using an evaluation scheme resulting in a evaluation result list (class \texttt{evaluationResultList}) with one entry (class \texttt{evaluationResult}) per algorithm. Each object of \texttt{evaluationResult} contains one or several object of \texttt{confusionMatrix} depending on the number of evaluations specified in the \texttt{evaluationScheme} (e.g., \(k\) for \(k\)-fold cross validation). With this infrastructure several recommender algorithms can be compared on a data set with a single line of code.

In the following, we will illustrate the usage of \texttt{recommenderlab} with several examples.

## 5. Examples

This fist few example shows how to manage data in recommender lab and then we create and evaluate recommenders. First, we load the package.
5.1. Coercion to and from rating matrices

For this example we create a small artificial data set as a matrix.

```
R> m <- matrix(sample(c(as.numeric(0:5), NA), 50,
+ replace=TRUE, prob=c(rep(.4/6,6),.6)), ncol=10,
+ dimnames=list(user=paste("u", 1:5, sep=''),
+ item=paste("i", 1:10, sep=''))
R> m
```

With coercion, the matrix can be easily converted into a realRatingMatrix object which stores the data in sparse format (only non-NA values are stored explicitly; NA values are represented by a dot).

```
R> r <- as(m, "realRatingMatrix")
R> r

5 x 10 rating matrix of class 'realRatingMatrix' with 19 ratings.

R> getRatingMatrix(r)

5 x 10 sparse Matrix of class "dgCMatrix"

u1 . 2 3 5 . 5 . 4 .
  u2 2 . . . . . . . 2 3
  u3 2 . . . 1 . . .
  u4 2 2 1 . 5 . 0 2 .
  u5 5 . . . . . . 5 . 4

The realRatingMatrix can be coerced back into a matrix which is identical to the original matrix.

```
R> identical(as(r, "matrix"),m)
[1] TRUE
```
It can also be coerced into a list of users with their ratings for closer inspection or into a data.frame with user/item/rating tuples.

```r
R> as(r, "list")

$u1
  i2 i3 i4 i6 i8
2 3 5 5 4

$u2
  i1 i9 i10
2 2 3

$u3
  i1 i6
2 1

$u4
  i1 i2 i3 i6 i8 i9
2 2 1 5 0 2

$u5
  i1 i8 i10
5 5 4

R> head(as(r, "data.frame"))

   user item rating
 5   u1   i2     2
 7   u1  i13    3
 9   u1  i14    5
10   u1  i16    5
13   u1  i18    4
 1   u2   i1     2
```

The data.frame version is especially suited for writing rating data to a file (e.g., by `write.csv()`). Coercion from data.frame (user/item/rating tuples) and list into a sparse rating matrix is also provided. This way, external rating data can easily be imported into `recommenderlab`.

### 5.2. Normalization

An important operation for rating matrices is to normalize the entries to, e.g., centering to remove rating bias by subtracting the row mean from all ratings in the row. This is can be easily done using `normalize()`.

```r
R> r_m <- normalize(r)
R> r_m
```
5 x 10 rating matrix of class ‘realRatingMatrix’ with 19 ratings.
Normalized using center on rows.

R> getRatingMatrix(r_m)
5 x 10 sparse Matrix of class "dgCMatrix"

u1 .   -1.8 -0.8 1.2 . 1.2 . 0.2000 . .
u2 -0.3333 . . . . . . . -0.3333 0.6667
u3 0.5000 . . . . -0.5 . . . .
u4 0.0000 0.0 -1.0 . . 3.0 -2.0000 0.0000 .
u5 0.3333 . . . . . . 0.3333 . -0.6667

Normalization can be reversed using denormalize().

R> denormalize(r_m)
5 x 10 rating matrix of class ‘realRatingMatrix’ with 19 ratings.

Small portions of rating matrices can be visually inspected using image().

R> image(r, main = "Raw Ratings")
R> image(r_m, main = "Normalized Ratings")

Figure 4 shows the resulting plots.

5.3. Binarization of data

A matrix with real valued ratings can be transformed into a 0-1 matrix with binarize() and a user specified threshold (min_ratings) on the raw or normalized ratings. In the following
only items with a rating of 4 or higher will become a positive rating in the new binary rating matrix.

\[
R> r_b <- \text{binarize}(r, \text{minRating}=4)
\]

\[
R> r_b
\]

5 x 10 rating matrix of class `binaryRatingMatrix` with 7 ratings.

\[
R> \text{as}(r_b, \text{"matrix"})
\]

5.4. Inspection of data set properties

We will use the data set Jester5k for the rest of this section. This data set comes with recommenderlab and contains a sample of 5000 users from the anonymous ratings data from the Jester Online Joke Recommender System collected between April 1999 and May 2003 (Goldberg, Roeder, Gupta, and Perkins 2001). The data set contains ratings for 100 jokes on a scale from −10 to +10. All users in the data set have rated 36 or more jokes.

\[
R> \text{data}(\text{Jester5k})
\]

\[
R> \text{Jester5k}
\]

5000 x 100 rating matrix of class `realRatingMatrix` with 362106 ratings.

Jester5k contains 362106 ratings. For the following examples we use only a subset of the data containing a sample of 1000 users (we set the random number generator seed for reproducibility). For random sampling sample() is provided for rating matrices.

\[
R> \text{set.seed}(1234)
\]

\[
R> r <- \text{sample}(\text{Jester5k}, 1000)
\]

\[
R> r
\]

1000 x 100 rating matrix of class `realRatingMatrix` with 71682 ratings.

This subset still contains 71682 ratings. Next, we inspect the ratings for the first user. We can select an individual user with the extraction operator.

\[
R> \text{rowCounts}(r[1,])
\]

u5092

70
The user has rated 70 jokes, the list shows the ratings and the user’s rating average is \(-2.413\).

Next, we look at several distributions to understand the data better. `getRatings()` extracts a vector with all non-missing ratings from a rating matrix.

In the histogram in Figure 5 shows an interesting distribution where all negative values occur with a almost identical frequency and the positive ratings more frequent with a steady decline towards the rating 10. Since this distribution can be the result of users with strong rating bias, we look next at the rating distribution after normalization.

Figure 6 shows that the distribution of ratings is closer to a normal distribution after row centering and Z-score normalization additionally reduces the variance to a range of roughly \(-3\) to \(+3\) standard deviations. It is interesting to see that there is a pronounced peak of ratings between zero and two.

Finally, we look at how many jokes each user has rated and what the mean rating for each joke is.
Histogram of $\text{getRatings}(r)$

**Figure 5:** Raw rating distribution for a sample of Jester.

Histogram of $\text{getRatings}(\text{normalize}(r))$

**Figure 6:** Histogram of normalized ratings using row centering (left) and Z-score normalization (right).
Figure 7: Distribution of the number of rated items per user (left) and of the average ratings per joke (right).

\[ \text{Histogram of } \text{rowCounts}(r) \]
\[ \text{Histogram of } \text{colMeans}(r) \]

\[ \text{Figure 7: Distribution of the number of rated items per user (left) and of the average ratings per joke (right).} \]

\[ R> \text{hist(colMeans}(r), \text{breaks}=20) \]

Figure 7 shows that there are unusually many users with ratings around 70 and users who have rated all jokes. The average ratings per joke look closer to a normal distribution with a mean above 0.

5.5. Creating a recommender

A recommender is created using the creator function `Recommender()`. Available recommendation methods are stored in a registry. The registry can be queried. Here we are only interested in methods for real-valued rating data.

\[ R> \text{recommenderRegistry$get\_entries(dataType = "realRatingMatrix")} \]

\$ALS\_realRatingMatrix
Recommender method: ALS for realRatingMatrix
Description: Recommender for explicit ratings based on latent factors, calculated by alternating least squares algorithm.
Parameters:
  - normalize lambda n_factors n_iterations min_item_nr seed
  1 NULL 0.1 10 10 1 NULL

\$ALS\_implicit\_realRatingMatrix
Recommender method: ALS_implicit for realRatingMatrix
Description: Recommender for implicit data based on latent factors, calculated by alternating least squares algorithm.
Parameters:
Developing and Testing Recommendation Algorithms

lambda alpha n_factors n_iterations min_item_nr seed
1 0.1 10 10 10 1 NULL

$IBCF_realRatingMatrix
Recommender method: IBCF for realRatingMatrix
Description: Recommender based on item-based collaborative filtering.
Reference: NA
Parameters:
  k method normalize normalize_sim_matrix alpha na_as_zero
1 30 "Cosine" "center" FALSE 0.5 FALSE

$POPULAR_realRatingMatrix
Recommender method: POPULAR for realRatingMatrix
Description: Recommender based on item popularity.
Reference: NA
Parameters:
  normalize
1 "center"

$RANDOM_realRatingMatrix
Recommender method: RANDOM for realRatingMatrix
Description: Produce random recommendations (real ratings).
Reference: NA
Parameters: None

$RERECOMMEND_realRatingMatrix
Recommender method: RERECOMMEND for realRatingMatrix
Description: Re-recommends highly rated items (real ratings).
Reference: NA
Parameters:
  randomize minRating
1 1 NA

$SVD_realRatingMatrix
Recommender method: SVD for realRatingMatrix
Description: Recommender based on SVD approximation with column-mean imputation.
Reference: NA
Parameters:
  k maxiter normalize
1 10 100 "center"

$SVDF_realRatingMatrix
Recommender method: SVDF for realRatingMatrix
Description: Recommender based on Funk SVD with gradient descend.
Reference: NA
Parameters:
k gamma lambda min_epochs max_epochs min_improvement normalize verbose
1 10 0.015 0.001 50 200 0.000001 "center" FALSE

$UBCF_realRatingMatrix
Recommender method: UBCF for realRatingMatrix
Description: Recommender based on user-based collaborative filtering.
Reference: NA
Parameters:
method nn sample normalize
1 "cosine" 25 FALSE "center"

Next, we create a recommender which generates recommendations solely on the popularity of items (the number of users who have the item in their profile). We create a recommender from the first 1000 users in the Jester5k data set.

R> r <- Recommender(Jester5k[1:1000], method = "POPULAR")
R> r

Recommender of type ‘POPULAR’ for ‘realRatingMatrix’
learned using 1000 users.

The model can be obtained from a recommender using getModel().

R> names(getModel(r))

[1] "topN"         "ratings"         "normalize"
[4] "aggregationRatings" "aggregationPopularity" "verbose"

R> getModel(r)$topN

Recommendations as ‘topNList’ with n = 100 for 1 users.

In this case the model has a top-N list to store the popularity order and further elements (average ratings, if it used normalization and the used aggregation function).

Recommendations are generated by predict() (consistent with its use for other types of models in R). The result are recommendations in the form of an object of class TopNList. Here we create top-5 recommendation lists for two users who were not used to learn the model.

R> recom <- predict(r, Jester5k[1001:1002], n=5)
R> recom

Recommendations as ‘topNList’ with n = 5 for 2 users.
The result contains two ordered top-$N$ recommendation lists, one for each user. The recommended items can be inspected as a list.

```
R> as(recom, "list")
```

```en
$u20089
[1] "j89" "j72" "j47" "j93" "j76"

$u11691
[1] "j89" "j93" "j76" "j88" "j96"
```

Since the top-$N$ lists are ordered, we can extract sublists of the best items in the top-$N$. For example, we can get the best 3 recommendations for each list using `bestN()`.

```
R> recom3 <- bestN(recom, n = 3)
R> recom3

Recommendations as ‘topNList’ with n = 3 for 2 users.
```

```
R> as(recom3, "list")
```

```en
$u20089
[1] "j89" "j72" "j47"

$u11691
[1] "j89" "j93" "j76"
```

Many recommender algorithms can also predict ratings. This is also implemented using `predict()` with the parameter `type` set to "ratings".

```
R> recom <- predict(r, Jester5k[1001:1002], type="ratings")
R> recom

2 x 100 rating matrix of class ‘realRatingMatrix’ with 97 ratings.
```

```
R> as(recom, "matrix")[,1:10]
```

```en
j1  j2  j3  j4  j5  j6  j7  j8  j9  j10
u20089 -1.013 -1.632 -1.8757 -3.685 NA NA NA -2.743 -0.8325
u11691  NA  NA -0.6377 -2.446 NA NA NA -1.505 NA
```

Predicted ratings are returned as an object of `realRatingMatrix`. The prediction contains `NA` for the items rated by the active users. In the example we show the predicted ratings for the first 10 items for the two active users.

Alternatively, we can also request the complete rating matrix which includes the original ratings by the user.
5.6. Evaluation of predicted ratings

Next, we will look at the evaluation of recommender algorithms. `recommenderlab` implements several standard evaluation methods for recommender systems. Evaluation starts with creating an evaluation scheme that determines what and how data is used for training and testing. Here we create an evaluation scheme which splits the first 1000 users in Jester5k into a training set (90%) and a test set (10%). For the test set 15 items will be given to the recommender algorithm and the other items will be held out for computing the error.

```r
R> e <- evaluationScheme(Jester5k[1:1000], method="split", train=0.9,
+   given=15, goodRating=5)
R> e
```

Evaluation scheme with 15 items given
Method: ‘split’ with 1 run(s).
Training set proportion: 0.900
Good ratings: >=5.000000
Data set: 1000 x 100 rating matrix of class ‘realRatingMatrix’ with 72358 ratings.

We create two recommenders (user-based and item-based collaborative filtering) using the training data.

```r
R> r1 <- Recommender(getData(e, "train"), "UBCF")
R> r1
```

Recommender of type ‘UBCF’ for ‘realRatingMatrix’ learned using 900 users.

```r
R> r2 <- Recommender(getData(e, "train"), "IBCF")
R> r2
```

Recommender of type ‘IBCF’ for ‘realRatingMatrix’ learned using 900 users.
Next, we compute predicted ratings for the known part of the test data (15 items for each user) using the two algorithms.

```r
R> p1 <- predict(r1, getData(e, "known"), type="ratings")
R> p1
100 x 100 rating matrix of class ‘realRatingMatrix’ with 8500 ratings.

R> p2 <- predict(r2, getData(e, "known"), type="ratings")
R> p2
100 x 100 rating matrix of class ‘realRatingMatrix’ with 8465 ratings.
```

Finally, we can calculate the error between the prediction and the unknown part of the test data.

```r
R> error <- rbind(
+   UBCF = calcPredictionAccuracy(p1, getData(e, "unknown")),
+   IBCF = calcPredictionAccuracy(p2, getData(e, "unknown"))
+ )
R> error
   RMSE  MSE  MAE
UBCF 4.402 19.38 3.463
IBCF 5.078 25.79 3.981
```

In this example user-based collaborative filtering produces a smaller prediction error.

### 5.7. Evaluation of a top-\(N\) recommender algorithm

For this example we create a 4-fold cross validation scheme with the the Given-3 protocol, i.e., for the test users all but three randomly selected items are withheld for evaluation.

```r
R> scheme <- evaluationScheme(Jester5k[1:1000], method="cross", k=4, given=3,
+   goodRating=5)
R> scheme
Evaluation scheme with 3 items given
Method: ‘cross-validation’ with 4 run(s).
Good ratings: >=5.000000
Data set: 1000 x 100 rating matrix of class ‘realRatingMatrix’ with 72358 ratings.
```

Next we use the created evaluation scheme to evaluate the recommender method popular. We evaluate top-1, top-3, top-5, top-10, top-15 and top-20 recommendation lists.

```r
R> results <- evaluate(scheme, method="POPULAR", type = "topNList",
+   n=c(1,3,5,10,15,20))
```
POPULAR run fold/sample [model time/prediction time]
  1 [0.005sec/0.32sec]
  2 [0.007sec/0.298sec]
  3 [0.006sec/0.282sec]
  4 [0.006sec/0.279sec]

R> results

Evaluation results for 4 folds/samples using method ‘POPULAR’.

The result is an object of class EvaluationResult which contains several confusion matrices. `getConfusionMatrix()` will return the confusion matrices for the 4 runs (we used 4-fold cross evaluation) as a list. In the following we look at the first element of the list which represents the first of the 4 runs.

R> getConfusionMatrix(results)[[1]]

```
   TP   FP   FN   TN  precision   recall  TPR   FPR
 1 0.488 0.512 16.016 79.988 0.4880 0.04039 0.04039 0.006031
 3 1.296 1.704 15.208 78.792 0.4320 0.11020 0.11020 0.020377
 5 2.084 2.916 14.420 77.580 0.4168 0.16709 0.16709 0.034883
10 4.052 5.948 12.452 74.551 0.4052 0.31215 0.31215 0.070909
15 5.736 9.264 10.768 71.231 0.3824 0.41957 0.41957 0.110526
20 7.128 12.872 9.376 67.620 0.3564 0.50119 0.50119 0.154023
```

For the first run we have 6 confusion matrices represented by rows, one for each of the six different top-N lists we used for evaluation. \( n \) is the number of recommendations per list. TP, FP, FN and TN are the entries for true positives, false positives, false negatives and true negatives in the confusion matrix. The remaining columns contain precomputed performance measures. The average for all runs can be obtained from the evaluation results directly using `avg()`.

R> avg(results)

```
   TP   FP   FN   TN  precision   recall  TPR   FPR
 1 0.451 0.549 16.721 79.279 0.4510 0.03703 0.03703 0.006505
 3 1.254 1.746 15.918 78.082 0.4180 0.09575 0.09575 0.020982
 5 2.032 2.968 15.136 76.864 0.4064 0.14672 0.14672 0.035639
10 3.926 6.074 13.250 73.751 0.3926 0.28082 0.28082 0.073155
15 5.649 9.351 11.520 70.481 0.3766 0.39295 0.39295 0.112699
20 7.042 12.958 10.129 66.871 0.3521 0.47678 0.47678 0.156597
```

Evaluation results can be plotted using `plot()`. The default plot is the ROC curve which plots the true positive rate (TPR) against the false positive rate (FPR).

R> plot(results, annotate=TRUE)
Figure 8: ROC curve for recommender method POPULAR.
For the plot where we annotated the curve with the size of the top-$N$ list is shown in Figure 8. By using "prec/rec" as the second argument, a precision-recall plot is produced (see Figure 9).

R> plot(results, "prec/rec", annotate=TRUE)

5.8. Comparing recommender algorithms

Comparing top-$N$ recommendations

The comparison of several recommender algorithms is one of the main functions of recommenderlab. For comparison also evaluate() is used. The only change is to use evaluate() with a list of algorithms together with their parameters instead of a single method name. In the following we use the evaluation scheme created above to compare the five recommender algorithms: random items, popular items, user-based CF, item-based CF, and SVD approximation. Note that when running the following code, the CF based algorithms are very slow.
Developing and Testing Recommendation Algorithms

For the evaluation we use a “all-but-5” scheme. This is indicated by a negative number for given.

R> set.seed(2016)
R> scheme <- evaluationScheme(Jester5k[1:1000], method="split", train = .9,
+ k=1, given=-5, goodRating=5)
R> scheme

Evaluation scheme using all-but-5 items
Method: ‘split’ with 1 run(s).
Training set proportion: 0.900
Good ratings: >=5.000000
Data set: 1000 x 100 rating matrix of class ‘realRatingMatrix’ with 72358 ratings.

R> algorithms <- list(
+ "random items" = list(name="RANDOM", param=NULL),
+ "popular items" = list(name="POPULAR", param=NULL),
+ "user-based CF" = list(name="UBCF", param=list(nn=50)),
+ "item-based CF" = list(name="IBCF", param=list(k=50)),
+ "SVD approximation" = list(name="SVD", param=list(k = 50))
+ )
R> ## run algorithms
R> results <- evaluate(scheme, algorithms, type = "topNList",
+ n=c(1, 3, 5, 10, 15, 20))

RANDOM run fold/sample [model time/prediction time]
  1 [0sec/0.018sec]
POPULAR run fold/sample [model time/prediction time]
  1 [0.005sec/0.113sec]
UBCF run fold/sample [model time/prediction time]
  1 [0.004sec/0.121sec]
IBCF run fold/sample [model time/prediction time]
  1 [0.035sec/0.016sec]
SVD run fold/sample [model time/prediction time]
  1 [0.099sec/0.013sec]

The result is an object of class evaluationResultList for the five recommender algorithms.

R> results

List of evaluation results for 5 recommenders:
Evaluation results for 1 folds/samples using method ‘RANDOM’.
Evaluation results for 1 folds/samples using method ‘POPULAR’.
Evaluation results for 1 folds/samples using method ‘UBCF’.
Evaluation results for 1 folds/samples using method ‘IBCF’.
Evaluation results for 1 folds/samples using method ‘SVD’.

Individual results can be accessed by list subsetting using an index or the name specified when calling `evaluate()`.

```r
R> names(results)

[1]  "random items"  "popular items"  "user-based CF"
[4]  "item-based CF"  "SVD approximation"
```

```r
R> results[["user-based CF"]]
```

Evaluation results for 1 folds/samples using method ‘UBCF’.

Again `plot()` can be used to create ROC and precision-recall plots (see Figures 10 and 11). Plot accepts most of the usual graphical parameters like `pch`, `type`, `lty`, etc. In addition `annotate` can be used to annotate the points on selected curves with the list length.

```r
R> plot(results, annotate=c(1,3), legend="bottomright")
```

```r
R> plot(results, "prec/rec", annotate=3, legend="topleft")
```

For this data set and the given evaluation scheme the user-based and item-based CF methods clearly outperform all other methods. In Figure 10 we see that they dominate the other method since for each length of top-N list they provide a better combination of TPR and FPR.

**Comparing ratings**

Next, we evaluate not top-N recommendations, but how well the algorithms can predict ratings.

```r
R> ## run algorithms
R> results <- evaluate(scheme, algorithms, type = "ratings")
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Fold/sample</th>
<th>Model time/prediction time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM</td>
<td>1</td>
<td>0.001sec/0.009sec</td>
</tr>
<tr>
<td>POPULAR</td>
<td>1</td>
<td>0.006sec/0.003sec</td>
</tr>
<tr>
<td>UBCF</td>
<td>1</td>
<td>0.004sec/0.113sec</td>
</tr>
<tr>
<td>IBCF</td>
<td>1</td>
<td>0.034sec/0.007sec</td>
</tr>
<tr>
<td>SVD</td>
<td>1</td>
<td>0.089sec/0.005sec</td>
</tr>
</tbody>
</table>

The result is again an object of class `evaluationResultList` for the five recommender algorithms.
Figure 10: Comparison of ROC curves for several recommender methods for the given-3 evaluation scheme.
Figure 11: Comparison of precision-recall curves for several recommender methods for the given-3 evaluation scheme.
Developing and Testing Recommendation Algorithms

Figure 12: Comparison of RMSE, MSE, and MAE for recommender methods for the given-3 evaluation scheme.

R> results

List of evaluation results for 5 recommenders:
Evaluation results for 1 folds/samples using method ‘RANDOM’.
Evaluation results for 1 folds/samples using method ‘POPULAR’.
Evaluation results for 1 folds/samples using method ‘UBCF’.
Evaluation results for 1 folds/samples using method ‘IBCF’.
Evaluation results for 1 folds/samples using method ‘SVD’.

R> plot(results, ylim = c(0,100))

Plotting the results shows a barplot with the root mean square error, the mean square error and the mean average error (see Figures 12).

Using a 0-1 data set

For comparison we will check how the algorithms compare given less information. We convert the data set into 0-1 data and instead of a all-but-5 we use the given-3 scheme.
Michael Hahsler

R> Jester_binary <- binarize(Jester5k, minRating=5)
R> Jester_binary <- Jester_binary[rowCounts(Jester_binary)>20]
R> Jester_binary

1797 x 100 rating matrix of class ‘binaryRatingMatrix’ with 65642 ratings.

R> scheme_binary <- evaluationScheme(Jester_binary[1:1000],
  + method="split", train=.9, k=1, given=3)
R> scheme_binary

Evaluation scheme with 3 items given
Method: ‘split’ with 1 run(s).
Training set proportion: 0.900
Good ratings: NA
Data set: 1000 x 100 rating matrix of class ‘binaryRatingMatrix’ with 36468 ratings.

R> results_binary <- evaluate(scheme_binary, algorithms,
  + type = "topNList", n=c(1,3,5,10,15,20))

RANDOM run fold/sample [model time/prediction time]
  1  [0.001sec/0.013sec]
POPULAR run fold/sample [model time/prediction time]
  1  [0.001sec/0.242sec]
UBCF run fold/sample [model time/prediction time]
  1  [0.001sec/0.323sec]
IBCF run fold/sample [model time/prediction time]
  1  [0.073sec/0.015sec]
SVD run fold/sample [model time/prediction time]
  1

Note that SVD does not implement a method for binary data and is thus skipped.

R> plot(results_binary, annotate=c(1,3), legend="topright")

From Figure 13 we see that given less information, the performance of item-based CF suffers
the most and the simple popularity based recommender performs almost as well as user-based
CF and association rules.

Similar to the examples presented here, it is easy to compare different recommender algo-
rithms for different data sets or to compare different algorithm settings (e.g., the influence of
neighborhood formation using different distance measures or different neighborhood sizes).

5.9. Implementing a new recommender algorithm

Adding a new recommender algorithm to recommenderlab is straight forward since it uses
a registry mechanism to manage the algorithms. To implement the actual recommender
algorithm we need to implement a creator function which takes a training data set, trains
Figure 13: Comparison of ROC curves for several recommender methods for the given-3 evaluation scheme.
a model and provides a predict function which uses the model to create recommendations for new data. The model and the predict function are both encapsulated in an object of class `Recommender`.

For examples look at the files starting with `RECOM` in the packages `R` directory. A good examples is in `RECOM_POPULAR.R`.

### 6. Conclusion

In this paper we described the `R` extension package `recommenderlab` which is especially geared towards developing and testing recommender algorithms. The package allows to create evaluation schemes following accepted methods and then use them to evaluate and compare recommender algorithms. `recommenderlab` currently includes several standard algorithms and adding new recommender algorithms to the package is facilitated by the built in registry mechanism to manage algorithms. In the future we will add more and more of these algorithms to the package and we hope that some algorithms will also be contributed by other researchers.

### Acknowledgments

This research was funded in part by the NSF Industry/University Cooperative Research Center for Net-Centric Software & Systems.

### References


Developing and Testing Recommendation Algorithms


**Affiliation:**

Michael Hahsler  
Engineering Management, Information, and Systems  
Lyle School of Engineering  
Southern Methodist University  
P.O. Box 750123  
Dallas, TX 75275-0123  
E-mail: mhahsler@lyle.smu.edu  
URL: http://lyle.smu.edu/~mhahsler