Developing and Testing Top-N Recommendation Algorithms for 0-1 Data using **recommenderlab** *

Michael Hahsler

February 27, 2011

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 computing recommendations using these type of data has direct application for example for large on-line retailers, there are many potential applications for recommender systems where such data is not available. However, in many cases there might be 0-1 rating data available or can be derived from other data sources (e.g., purchase records, Web click data) which can be utilized. Although this type of data differs significantly from directly elicited ratings, only very limited research is available for 0-1 data. This paper describes **recommenderlab** which provides the infrastructure to test and develop recommender algorithms. Currently the focus is on recommender systems for 0-1 data, however in the future it can be extended to also the more thoroughly researched case of directly elicited, real-valued rating data.

1 Introduction

Predicting ratings and creating personalized recommendations for products like books, songs or movies online came a long way from 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.com, iTunes and Netflix). 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 [1], 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 just awarded to the Belcore Pragmatic Chaos team.

The most widely used method to create recommendations is *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). For these type of recommender systems, several projects were initiated to implement recommender algorithms

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

[1] `http://www.netflixprize.com/`

(e.g., Apache Mahout/Taste [2], Cofi [3], RACOFI [4], SUGGEST [5], Vogoo PHP [6]).

Very limited 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 to 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 web site 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.

The R extension package **recommenderlab** provides a general infrastructure for collaborative filtering. In this paper we will focus on the package's capabilities for creating and testing recommender algorithms which create top-N recommendation list for 0-1 data.

This paper is structured as follows. Section 2 introduces collaborative filtering and applies popular methods to the top-N recommendation problem on 0-1 data. 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.

2 Collaborative Filtering for 0-1 Data

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 $\mathcal{U} = \{u_1, u_2, \ldots, u_m\}$ and a set of items $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$. Ratings are stored in a $m \times n$ rating matrix $\mathbf{R} = (r_{jk})$ where each row represent a user u_j with $1 \geq j \geq m$ and columns represent items i_k with $1 \geq k \geq n$. 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 $\mathbf{R}$ the values are missing. Most published 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., $[1, 5]$). However in this paper we concentrate on the 0-1 case with $r_{jk} \in 0, 1$ where we define:

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

Pan et al. (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. Two strategies

[2] `http://lucene.apache.org/mahout/`
[3] `http://www.nongnu.org/cofi/`
[4] `http://racofi.elg.ca/`
[5] `http://glaros.dtc.umn.edu/gkhome/suggest/overview/`
[6] `http://www.vogoo-api.com/`

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. Here will will follow mostly the first strategy based on the assumption that users typically favor only a small fraction of the items and thus most items with no rating will be indeed negative examples. However, it has to be said that Pan et al. (2008) propose strategies which represent a trade-off between the two extreme strategies based on wighted low rank approximations of the rating matrix and on negative example sampling which might improve results across all recommender algorithms. This is however outside the scope of this paper.

The aim of collaborative filtering is to create recommendations for a user called the active user $u_a \in \mathcal{U}$. We define the set of items unknown to user u_a as $\mathcal{I}_a = \mathcal{I} \setminus \{i_l \in \mathcal{I} | r_{al} = 1\}$. The two typical tasks are to predict ratings for all items in $\mathcal{I}_a$ or to create a list of the best N recommendations (i.e., a top-N recommendation list) for u_a (Sarwar, Karypis, Konstan, and Riedl, 2001). Creating a top-N lists can be seen as a second step after predicting ratings for all unknown items in $\mathcal{I}_a$ and then taking the N items with the highest predicted ratings. 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.

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 $\mathcal{I}_a$ (zeros in the 0-1 case) are replaced by ratings estimated from other data in $\mathbf{R}$. The estimated ratings can either be in $\{0, 1\}$ or in $[0, 1]$, where estimates closer to 1 indicate a stronger recommendation. The latter type of estimation allows for ordering and thus is needed to be able to create a top-N list. A list of top-N recommendations for a user u_a is an partially ordered set $T_N = (\mathcal{X}, \geq)$, where $\mathcal{X} \subset \mathcal{I}_a$ and $|\mathcal{X}| \leq N$ ($|\cdot|$ 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 $|\mathcal{I}_a| < N$ 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 $x, y \in \mathcal{X}$. Furthermore we require that $\forall_{x \in \mathcal{X}} \forall_{y \in \mathcal{I}_a} \quad \hat{r}_{ax} \geq \hat{r}_{ay}$ to ensure that the top-N list contains only the items with the highest estimated rating.

Collaborative filtering algorithms are typically divided into two groups, *memory-based* and *model-based* algorithms (Breese, Heckerman, and Kadie, 1998). Memory-based algorithms 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 well known memory and model-based collaborative filtering algorithms and apply them to 0-1 data.

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 based on analysis of rating data. The assumption is that users with similar preferences will rate products 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

$$\mathrm{sim}_{\mathrm{Pearson}}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i \in I} (\mathbf{x}_i \, \bar{\mathbf{x}})(\mathbf{y}_i \, \bar{\mathbf{y}})}{(|I| - 1) \, \mathrm{sd}(\mathbf{x}) \, \mathrm{sd}(\mathbf{y})}$$

and

$$\mathrm{sim}_{\mathrm{Cosine}}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2},$$

where $\mathbf{x} = r_{x\cdot}$ and $\mathbf{y} = r_{y\cdot}$ represent the users' profile vectors. $\mathrm{sd}(\cdot)$ is the standard deviation and $\|\cdot\|_2$ is the l^2-norm. For calculating similarity using rating data only the dimensions

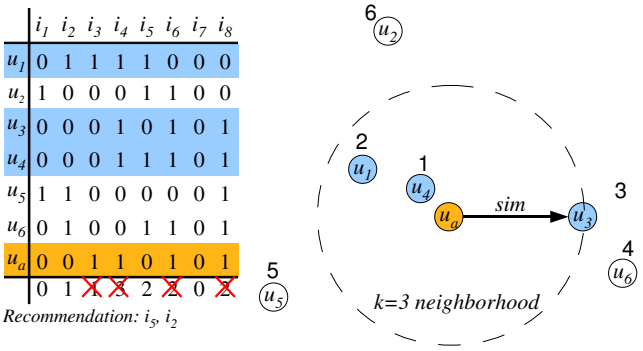

Figure 1: User-based collaborative filtering.

(items) are used which were rated by both users. However, for 0-1 data that would lead to the problem that the vectors $\mathbf{x}$ and $\mathbf{y}$ only contain ones and thus no useful measure can be calculated.

If we assume that most zeroes are actually items that the user does not like, we can use all items in the similarity calculation. However, this will produce significant errors for newer users with very few ones. A similarity measure which only focuses on matching ones and thus prevents the problem with zeroes is the *Jaccard index*:

$$\text{sim}_{\text{Jaccard}}(\mathcal{X}, \mathcal{Y}) = \frac{|\mathcal{X} \cap \mathcal{Y}|}{|\mathcal{X} \cup \mathcal{Y}|},$$

where $\mathcal{X}$ and $\mathcal{Y}$ are the sets of the items with a 1 in user profiles u_a and u_b, respectively.

Now the neighborhood $\mathcal{N} \subset \mathcal{U}$ 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. For real valued ratings, Breese et al. (1998) suggest to aggregate the ratings for item i_j as

$$\hat{r}_{aj} = \bar{r}_a + \kappa \sum_{i \in \mathcal{N}} w_{ai}(r_{ij} - \bar{r}_i)$$

where $\bar{r}_x$ is the mean of the ratings of user u_x, w_{ai} is the weight for user u_i and κ is a normalizing factor to make the weights sum to 1. The weights can reflect the similarity between the user and the active user. For 0-1 data we suggest, following Weiss and Indurkhya (2001), to calculate the following score.

$$s_{aj} = \sum_{i \in \mathcal{N}} w_{ai}\, r_{ij}$$

This score is not normalized but can be easily used to find the top-N items with the highest score.

An example of the process of creating recommendations for 0-1 data by user-based CF is shown in Figure 1. To the left is the rating matrix $\mathbf{R}$ with 6 users and 8 items. The active user u_a we want to create recommendations for is 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 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 closer) with the active user in the center. The $k = 3$ nearest neighbors are selected and marked in the database to the left. To generate an aggregated score, we use for the example a weight of 1 for all users. Thus the ones in the selected users are just summed up. Then items known to the active user are removed and the N items with the highest score (greater than zero) form the top-N recommendations. In the example in Figure 1 only two items are recommended.

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.

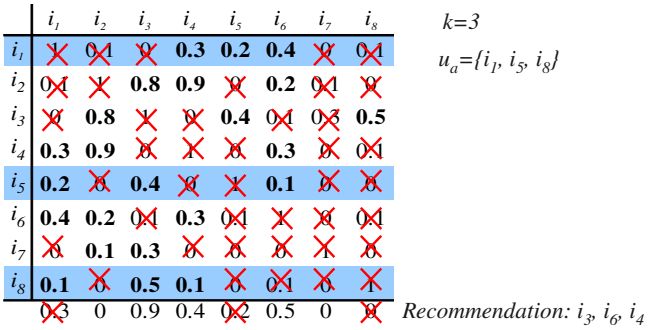

Figure 2: Item-based collaborative filtering

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 the 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. For 0-1 data again the Jaccard index can be used giving focus to matching ones. For item-based CF, Deshpande and Karypis (2004) proposed a *Conditional probability-based similarity* defined as:

$$\text{sim}_{\text{Conditional}}(x, y) = \frac{\text{Freq}(xy)}{\text{Freq}(x)} = \hat{P}(y|x),$$

where $x, y \in \mathcal{I}$ are two items and $\text{Freq}(\cdot)$ is the number of users with the given items in their profile. This similarity is in fact an estimate of the conditional probability to see item y in a profile given that the profile contains items x. This similarity is equivalent to the confidence measure used for association rules (see section 2.3 below). A drawback of this similarity measure is its sensitivity to the frequency of x with rare x producing high similarities. To reduce the sensitivity, Deshpande and Karypis (2004) propose a normalized version of the similarity measure:

$$\text{sim}_{\text{Karypis}}(x, y) = \frac{\sum_{\forall i b_{i,x}} b_{i,y}}{\text{Freq}(x)\text{Freq}(y)^{\alpha}}$$

where $\mathbf{B} = (b_{i,j})$ is a normalized rating matrix where all rows sum up to 1. $\text{Freq}(y)^{\alpha}$ reduces the problem with rare x.

All pairwise similarities are stored in a $N \times N$ similarity matrix $\mathbf{S}$, which is again normalized such that rows sum up to 1. 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 (Sarwar et al., 2001). This can improve the space and time complexity significantly by sacrificing some recommendation quality.

Figure 2 shows an example for $N = 8$ items. For the normalized similarity matrix $\mathbf{S}$ only $k = 3$ entries are stored per row (the crossed out entries are discarded).

To make a recommendation based on the model only two steps are necessary:

1. Calculate a score for each item by adding the similarities with the active user's items.
2. Remove the items of the active user and recommend the N items with the highest score.

In Figure 2 we assume that the active user prefers items i_1, i_5 and i_8. The rows corresponding to these items are highlighted and the rows are added up. Now the sums for the items preferred by the user are removed (crossed out in Figure 2), leaving three items with a score larger than zero which results in the top-N recommendation list i_3, i_6, i_4.

Item-based CF are very efficient since the models (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 Association Rules

Recommender systems based on association rules produce recommendations based on a dependency 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 $\mathcal{T}_k = \{i_j \in \mathcal{I} | r_{jk} = 1\}$ and the whole transaction data base is $\mathcal{D} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_U\}$ 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:

$$\text{support}(\mathcal{X} \rightarrow \mathcal{Y}) = \text{support}(\mathcal{X} \cup \mathcal{Y}) = \text{Freq}(\mathcal{X} \cup \mathcal{Y})/|\mathcal{D}|$$

$$\text{confidence}(\mathcal{X} \rightarrow \mathcal{Y}) = \text{support}(\mathcal{X} \cup \mathcal{Y})/\text{support}(\mathcal{X}) = \hat{P}(\mathcal{Y}|\mathcal{X})$$

$\text{Freq}(\mathcal{X})$ gives the number of transactions in the data base $\mathcal{D}$ that contains all items in $\mathcal{X}$.

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 confidence (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 ($\mathcal{X}$), it incorporates higher order effects between more than two items.

3 Evaluation of Top-N Recommender Algorithms for 0-1 Data

Evaluation of recommender systems is an important topic but most evaluation efforts center on recommender systems for non-binary rating data. A comprehensive review was presented by Herlocker, Konstan, Terveen, and Riedl (2004). However, here we will discuss the evaluation of top-N recommender algorithms for 0-1 data.

Typically, given a rating matrix $\mathbf{R}$, recommender algorithms are evaluated by first partitioning the users (rows) in $\mathbf{R}$ into two sets $\mathcal{U}_{train} \cup \mathcal{U}_{test} = \mathcal{U}$. The rows of $\mathbf{R}$ corresponding to the training users U_{train} are used to learn the recommender model. Then each user $u_a \in \mathcal{U}_{test}$ is seen as an active user, however, before creating recommendations some items are withheld from the profile $r_{u_a \cdot}$ and it measured how well these removed items are predicted by the recommender algorithm in its top-N list. This type of evaluation does not take into account that 0 in the data also codes for items that are unknown to the user and could potentially be a good recommendation. However, it is assumed that if a recommender algorithm performed better

Table 1: 2x2 confusion matrix

actual / predicted	**negative**	**positive**
negative	a	b
positive	c	d

in predicting the withheld items, it will also perform better in finding good recommendations for unknown items.

To determine how to split $\mathcal{U}$ into $\mathcal{U}_{train}$ and $\mathcal{U}_{test}$ 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 $\mathcal{U}_{test}$ 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 $\mathcal{U}$ 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.

The prediction results for all test users $\mathcal{U}_{test}$ can be aggregated into a so called *confusion matrix* depicted in table 1 (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.

$$Accuracy = \frac{correct\ recommendations}{total\ possible\ recommendations} = \frac{a+d}{a+b+c+d} \tag{2}$$

A common error measure is the *mean absolute error* (*MAE*, also called *mean absolute deviation* or *MAD*).

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|\epsilon_i| = \frac{b+c}{a+b+c+d}, \tag{3}$$

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

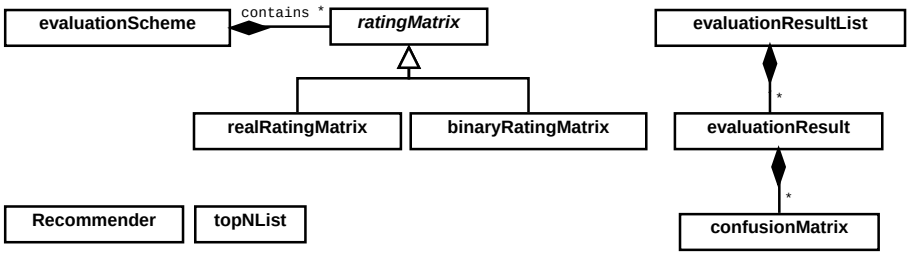

Figure 3: UML class diagram for package **recommenderlab** (Fowler, 2004).

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).

$$Precision = \frac{correctly\ recommended\ items}{total\ recommended\ items} = \frac{d}{b+d} \tag{4}$$

$$Recall = \frac{correctly\ recommended\ items}{total\ useful\ recommendations} = \frac{d}{c+d} \tag{5}$$

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 α controls the trade-off between precision and recall.

$$E\text{-}measure = \frac{1}{\alpha(1/Precision) + (1-\alpha)(1/Recall)} \tag{6}$$

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

$$F\text{-}measure = \frac{2\,Precision\,Recall}{Precision + Recall} = \frac{2}{1/Precision + 1/Recall} \tag{7}$$

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 5) by the *probability of false alarm* (also called false positive rate FPR or $1 - specificity$, 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 the rating matrix $\mathbf{R}$. 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**. Although the focus of the package and this paper is on 0-1 data, for completeness the package contains class realRatingMatrix which implements a rating matrix with real valued ratings stored in sparse format defined in package **Matrix**. With this class the infrastructure of **recommenderlab** can be easily extended to non-binary data.

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, ...).
```

`object` is the recommender object, `newdata` is the data for the active users and `n` is the number of recommended items N in each top-N list. `predict()` will return a list of objects of class topNList, one for each of the active users in `newdata`.

The actual implementations for the recommendation algorithms are managed using the registry mechanism provided by package **registry**. Generally, the registry mechanism is hidden from the user and the creator function `Recommender()` uses it in the background to map a recommender method name to its implementation. However, the registry can be directly queried 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 **recommenderlab** provides the infrastructure to create and maintain evaluation schemes stored as an object of class evaluationScheme from rating data. The creator function

```
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) with item withholding (parameter `given`). The function `evaluate()` is then used to evaluate several recommender algorithms using an evaluation scheme resulting in a evaluation result list (class evaluationResultList) with one entry (class evaluationResult) per algorithm. Each object of evaluationResult contains one or several object of confusionMatrix depending on the number of evaluations specified in the 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 **recommenderlab** with several examples.

5 Examples

5.1 A first session

This fist example shows how to train and use a recommender algorithm. For the example we use the data set *MSWeb* which is included in **recommenderlab**. The data set contains anonymous web click-stream data from `www.microsoft.com` for 38,000 anonymous, randomly selected users. For each user, the data lists all the areas of the web site (called Vroots; e.g., MS Office, Windows NT Server) that the user visited in a one week time frame (Breese et al., 1998).

We first load the package and the data set and then select for the example only users who visited more than 5 areas. Since the areas/items are columns in the 0-1 rating matrix, we select all rows/users with a row count larger than 5.

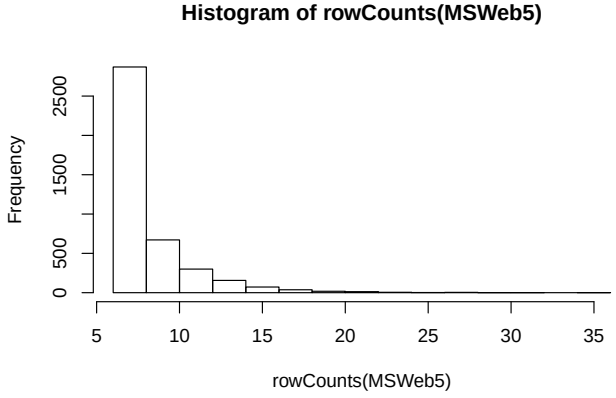

Figure 4: Distribution of the number of items per user (rowCounts).

```
R> library("recommenderlab")
R> data(MSWeb)
R> MSWeb5 <- MSWeb[rowCounts(MSWeb) > 5, ]
R> MSWeb5

4151 x 285 rating matrix of class 'binaryRatingMatrix'
with 33875 ratings.
```

The used data set contains 4151 users with more than 5 items and 285 items. To understand the distribution of the number of areas/items each user visited, we can produce a histogram.

```
R> hist(rowCounts(MSWeb5), breaks = 20)
```

Since the rows in the rating matrix represent the users, the histogram in Figure 4 shows the distribution of the users with $6, 7, \ldots$ items (note that we required users to have at least 5 items). On average a user has 8.16 items in her/his profile.

The profile of users can be inspected with `LIST()`. For example, we can see what areas the first two users visited.

```
R> LIST(MSWeb5[1:2])

$`10`
[1] "regwiz"                "Visual Basic"          "MS Office Development"
[4] "Outlook Development"   "Visual Basic Support"  "Office Free Stuff"

$`19`
 [1] "End User Produced View"    "Knowledge Base"
 [3] "Microsoft.com Search"      "Free Downloads"
 [5] "Products"                  "isapi"
 [7] "Clip Gallery Live"         "Windows NT Server"
 [9] "MS Office"                 "Games"
[11] "MS Store Logo Merchandise"
```

It is also interesting to study the popularity of items. To get a first idea, we use an image plot for 200 randomly chosen users.

```
R> image(sample(MSWeb5, 200))
```

Figure 5 shows that some items are by far more popular than others. We can further look at the distribution of how many user profiles contain an item. This is done by plotting a histogram for the column sums of the rating matrix.

```
R> hist(colCounts(MSWeb5), breaks = 25)
```

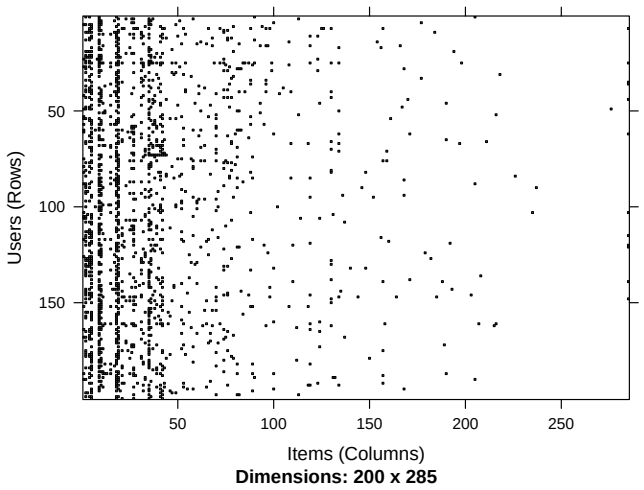

Figure 5: Image plot of 200 randomly chosen users in the rating matrix **R**. Dark squares represent 1s in the matrix.

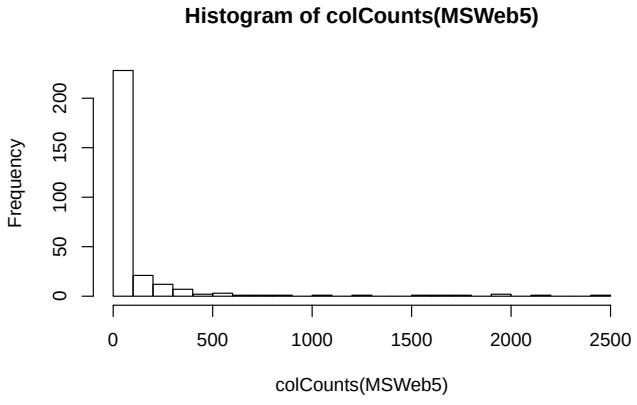

Figure 6: Distribution of item popularity (colCounts).

The histogram is shown in Figure 6. We see a typical distribution where the number of items falls quickly with popularity and only very few items are extremely popular (tail of the distribution).

A recommender is created using the creator function `recommender()`. Here we create a simple 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 using the first 1000 users in the data set.

```
R> r <- Recommender(MSWeb5[1:1000], method = "POPULAR")
R> r
```

```
Recommender of type 'POPULAR' for 'binaryRatingMatrix' learned using 1000 users.
```

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

```
R> getModel(r)
```

```
$description
[1] "Order of items by popularity"

$popOrder
  [1]   9  19  18   5  35   2  10   4  27  36  42  41  38  31  39  21  26
 [18]  33   1  37  52  11  32  70  15  74  28   3  53  78  47  25  76  58
 [35]  67  60 130 285  64  65  43  49 119  17   8  54  22  46  40  81  88
 [52]  12  23  68 134  87  57  82  44  72  77  48  56  69  75  61  89   7
 [69]  55  59 118  51 113 125 137  71  85  95 136  20  29  63 123 124  96
 [86]  62  84 105 167  83  90 157 168  66  93 150  79  99 135 146 154  24
[103]  45 100 189  14  16 127 143 204  50  92 109  13  30  80  97 102 108
[120] 114 131 140 141 148 158 159 190  91 110 112 121 133 144 156 164 169
[137] 183 147 160 177 188 201 203  94  98 103 126 152 155 162 176 181 184
[154] 193 197 205 206 218  34  86 101 111 166 170 172 174 185 200 211 216
[171] 227   6 104 106 115 132 139 151 161 163 165 179 182 187 192 194 198
[188] 202 207 215 219 220 222 223 226 230 231 234 241 107 116 120 128 129
[205] 138 145 149 153 171 173 178 186 191 195 196 199 208 210 212 221 225
[222] 228 236 237 238 240 244  73 117 122 142 175 180 209 213 214 217 224
[239] 229 232 233 235 239 242 243 245 246 247 248 249 250 251 252 253 254
[256] 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
[273] 272 273 274 275 276 277 278 279 280 281 282 283 284
```

In this case the model is just a short description and a simple vector called `popOrder` containing the order of items according to popularity in the data set.

Recommendations are generated by `predict()` in the same way predict is used in R for other types of models. 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, MSWeb5[1001:1002], n = 5)
R> recom
```

```
Recommendations as 'topNList' with n = 5 for 2 users.
```

The result are two ordered top-N recommendation lists, one for each user. The recommended items can be inspected using `LIST()`.

```
R> LIST(recom)
```

```
[[1]]
[1] "Free Downloads"    "Products"          "Internet Explorer"
[4] "Support Desktop"   "Knowledge Base"

[[2]]
[1] "isapi"
[2] "Microsoft.com Search"
[3] "Support Desktop"
```

```
[4] "Knowledge Base"
[5] "Internet Site Construction for Developers"
```

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> LIST(recom3)

[[1]]
[1] "Free Downloads"    "Products"          "Internet Explorer"

[[2]]
[1] "isapi"                "Microsoft.com Search" "Support Desktop"
```

Next we will look at the evaluation of recommender algorithms.

5.2 Evaluation of a recommender algorithm

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 evaluation. Here 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> scheme <- evaluationScheme(MSWeb5, method = "cross", k = 4,
+     given = 3)
R> scheme

Evaluation scheme with 3 items given
Method: 'cross-validation' with 4 runs (training set proportion: NA)
Data set: 4151 x 285 rating matrix of class 'binaryRatingMatrix'
with 33875 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> results <- evaluate(scheme, method = "POPULAR", n = c(1,
+     3, 5, 10, 15, 20))

POPULAR run 1 [1.38 s] 2 [1.344 s] 3 [1.476 s] 4 [1.356 s]

R> results

Evaluation results for 4 runs 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]]

n        TP     FP     FN    TN     PP  recall precision      FPR     TPR
  1   1.568  2.081 17.298 264.1  3.649 0.08313    0.4298 0.007818 0.08313
  3   4.130  6.818 14.737 259.3 10.947 0.21890    0.3772 0.025617 0.21890
  5   6.432 11.814 12.435 254.3 18.246 0.34090    0.3525 0.044391 0.34090
  10  9.512 26.979  9.354 239.2 36.491 0.50418    0.2607 0.101374 0.50418
  15 11.116 43.621  7.751 222.5 54.737 0.58918    0.2031 0.163907 0.58918
  20 12.102 60.881  6.765 205.3 72.982 0.64144    0.1658 0.228760 0.64144
```

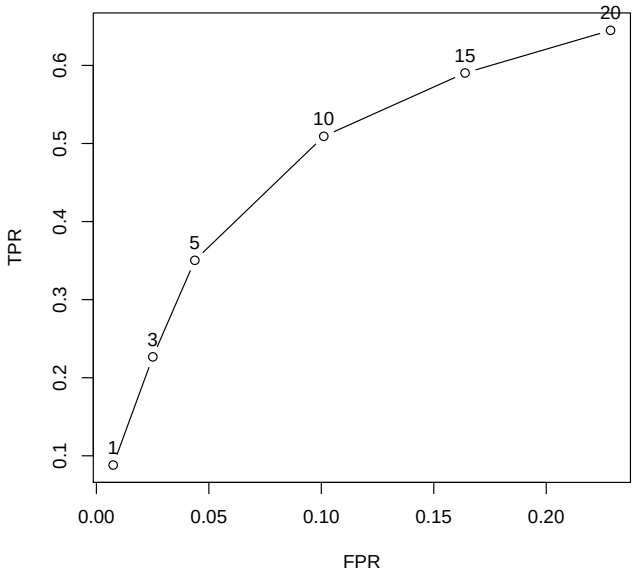

Figure 7: ROC curve for recommender method POPULAR.

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)
```

```
n        TP     FP     FN    TN     PP  recall precision      FPR     TPR
  1   1.660  1.989 17.161 264.2  3.649 0.08818    0.4548 0.007474 0.08818
  3   4.268  6.680 14.553 259.5 10.947 0.22674    0.3898 0.025095 0.22674
  5   6.595 11.651 12.225 254.5 18.246 0.35038    0.3614 0.043770 0.35038
  10  9.583 26.908  9.237 239.3 36.491 0.50920    0.2626 0.101089 0.50920
  15 11.111 43.626  7.710 222.6 54.737 0.59033    0.2030 0.163898 0.59033
  20 12.138 60.845  6.682 205.3 72.982 0.64492    0.1663 0.228585 0.64492
```

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)
```

For the plot where we annotated the curve with the size of the top-N list is shown in Figure 7. By using `"prec/rec"` as the second argument, a precision-recall plot is produced (see Figure 8).

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

5.3 Comparing recommender algorithms

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

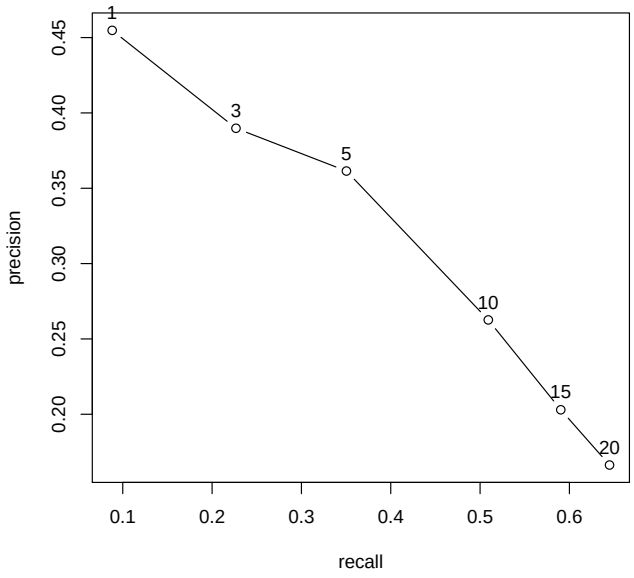

Figure 8: Precision-recall plot for method POPULAR.

algorithms: random items, popular items, user-based CF, item-based CF, and association rule based recommendations. Note that when running the following code, the CF based algorithms are very slow.

```
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(method = "Jaccard", nn = 50)), `item-based CF` = list(name = "IBCF",
+     param = list(method = "Jaccard", k = 50)), `association rules` = list(name = "AR",
+     param = list(supp = 0.001, conf = 0.2, maxlen = 2)))
R> results <- evaluate(scheme, algorithms, n = c(1, 3, 5, 10,
+     15, 20))
```

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 4 runs using method 'RANDOM'.
Evaluation results for 4 runs using method 'POPULAR'.
Evaluation results for 4 runs using method 'UBCF'.
Evaluation results for 4 runs using method 'IBCF'.
Evaluation results for 4 runs using method 'AR'.
```

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

```
R> names(results)

[1] "random items"      "popular items"     "user-based CF"
[4] "item-based CF"     "association rules"

R> results[["association rules"]]

Evaluation results for 4 runs using method 'AR'.
```

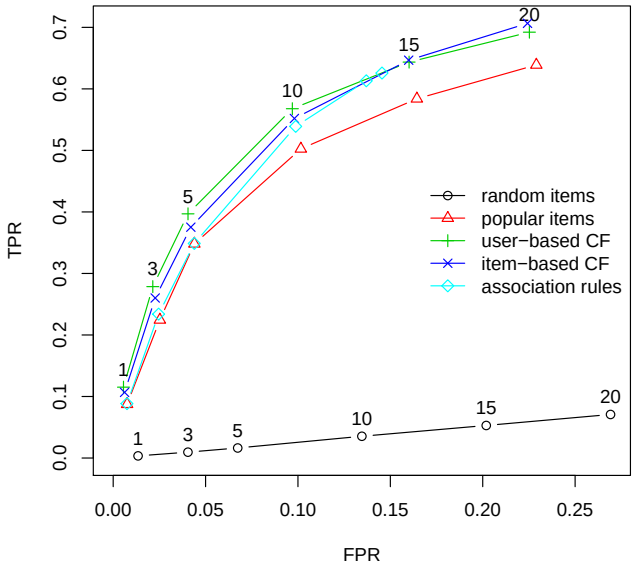

Figure 9: Comparison of ROC curves for several recommender methods for the given-3 evaluation scheme.

Again `plot()` can be used to create ROC and precision-recall plots (see Figures 9 and 10). 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> plot(results, annotate = c(1, 3), legend = "right")

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

For this data set and the given evaluation scheme the user-based and item-based CF methods clearly outperform all other methods. In Figure 9 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.

For comparison we will check how the algorithms compare given less information using instead of a given-3 a given-1 scheme.

```
R> scheme1 <- evaluationScheme(MSWeb5, method = "cross", k = 4,
+     given = 1)
R> scheme1

Evaluation scheme with 1 items given
Method: 'cross-validation' with 4 runs (training set proportion: NA)
Data set: 4151 x 285 rating matrix of class 'binaryRatingMatrix'
with 33875 ratings.

R> results1 <- evaluate(scheme1, algorithms, n = c(1, 3, 5,
+     10, 15, 20))

R> plot(results1, annotate = c(1, 3), legend = "right")
```

From Figure 11 we see that given less information, the performance of item-based CF suffers the most and the simple popularity based recommender performs almost a well as user-based CF and association rules.

Similar to the examples presented here, it is easy to compare different recommender algorithms 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).

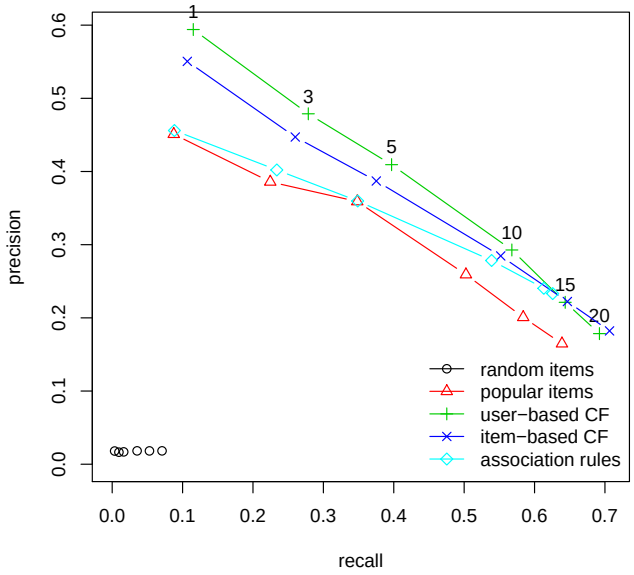

Figure 10: Comparison of precision-recall curves for several recommender methods for the given-3 evaluation scheme.

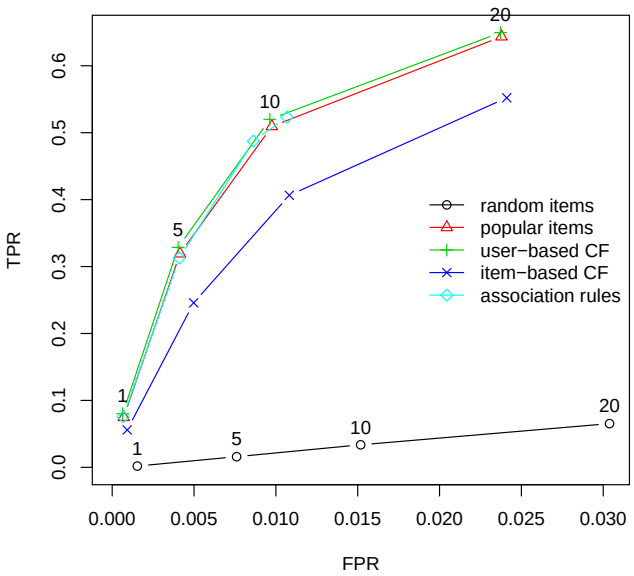

Figure 11: Comparison of ROC curves for several recommender methods for the given-1 evaluation scheme.

Table 2: Defining and registering a new recommender algorithm.

```
1  ## always recommends the top-N popular items (without known items)
2  BIN_POPULAR <- function(data, parameter = NULL) {
3
4    model <- list(
5      description = "Order of items by popularity",
6      popOrder = order(colCounts(data), decreasing=TRUE)
7    )
8
9    predict <- function(model, newdata, n=10) {
10     n <- as.integer(n)
11
12     ## remove known items and take highest
13     reclist <- lapply(LIST(newdata, decode= FALSE),
14       function(x) head(model$popOrder[!(model$popOrder %in% x)], n))
15
16     new("topNList", items = reclist, itemLabels = colnames(newdata), n = n)
17   }
18
19   ## construct recommender object
20   new("Recommender", method = "POPULAR", dataType = "binaryRatingMatrix",
21     ntrain = nrow(data), model = model, predict = predict)
22 }
23
24 ## register recommender
25 recommenderRegistry$set_entry(
26   method="POPULAR", dataType = "binaryRatingMatrix", fun=BIN_POPULAR,
27   description="Recommender based on item popularity (binary data)."
28 )
```

5.4 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 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 example the creator function in Table 2 is called `BIN_POPULAR()`. It uses the (training) data to create a model which is a simple list (lines 4–7 in Table 2). In this case the model is just a list of all items sorted in decreasing order of popularity. The second part (lines 9–22) is the predict function which takes the model, new data and the number of items of the desired top-N list as its arguments. Predict used the model to compute recommendations for each user in the new data and encodes them as an object of class topNList (line 16). Finally, the trained model and the predict function are returned as an object of class Recommender (lines 20–21). Now all that needs to be done is to register the creator function. In this case it is called POPULAR and applies to binary rating data (lines 25–28).

To create a new recommender algorithm the code in Table 2 can be copied. Then lines 5, 6, 20, 26 and 27 need to be edited to reflect the new method name and description. Line 6 needs to be replaced by the new model. More complicated models might use several entries in the list. Finally, lines 12–14 need to be replaced by the recommendation code.

6 Conclusion

Being able to use automated recommender systems offers a big advantage for online retailers and for many other applications. But often no extensive data base of rating data is available and it makes sense to think about using 0-1 data, which is in many cases easier to obtain, instead. Unfortunately there is only limited research on collaborative filtering based recommender systems using 0-1 data available.

In this paper we described the R extension package **recommenderlab** which is especially geared towards developing and testing recommender algorithms for 0-1 data. The package allows to create evaluation schemes following accepted methods and then use them to evaluate and compare recommender algorithms. Adding new recommender algorithms to the package is facilitates using a registry to manage the algorithms.

recommenderlab currently includes several algorithms for 0-1 data, however, the infrastructure is flexible enough to also extend to the more conventional non-binary rating data and its 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.

References

R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499, Santiago, Chile, September 1994.

J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Uncertainty in Artificial Intelligence. Proceedings of the Fourteenth Conference*, pages 43–52, 1998.

A. Demiriz. Enhancing product recommender systems on sparse binary data. *Data Minining and Knowledge Discovery*, 9(2):147–170, 2004. ISSN 1384-5810.

M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems*, 22(1):143–177, 2004. ISSN 1046-8188.

M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, third edition, 2004.

X. Fu, J. Budzik, and K. J. Hammond. Mining navigation history for recommendation. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 106–112. ACM, 2000. ISBN 1-58113-134-8.

A. Geyer-Schulz, M. Hahsler, and M. Jahn. A customer purchase incidence model applied to recommender systems. In R. Kohavi, B. Masand, M. Spiliopoulou, and J. Srivastava, editors, *WEBKDD 2001 - Mining Log Data Across All Customer Touch Points, Third International Workshop, San Francisco, CA, USA, August 26, 2001, Revised Papers*, Lecture Notes in Computer Science LNAI 2356, pages 25–47. Springer-Verlag, July 2002.

D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/138859.138867.

J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation. *Data Mining and Knowledge Discovery*, 8:53–87, 2004.

J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, 22(1):5–53, January 2004. ISSN 1046-8188. doi: 10.1145/963770.963772.

B. Kitts, D. Freed, and M. Vrieze. Cross-sell: a fast promotion-tunable customer-item recommendation method based on conditionally independent probabilities. In *KDD '00:*

Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 437–446. ACM, 2000. ISBN 1-58113-233-6. doi: http://doi.acm.org/10.1145/347090.347181.

R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1137–1143, 1995.

R. Kohavi and F. Provost. Glossary of terms. *Machine Learning*, 30(2–3):271–274, 1998.

J.-S. Lee, C.-H. Jun, J. Lee, and S. Kim. Classification-based collaborative filtering using market basket data. *Expert Systems with Applications*, 29(3):700–704, October 2005.

W. Lin, S. A. Alvarez, and C. Ruiz. Efficient adaptive-support association rule mining for recommender systems. *Data Mining and Knowledge Discovery*, 6(1):83–105, 2002. ISSN 1384-5810.

G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

T. W. Malone, K. R. Grant, F. A. Turbak, S. A. Brobst, and M. D. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5):390–402, 1987. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/22899.22903.

A. Mild and T. Reutterer. An improved collaborative filtering approach for predicting cross-category purchases based on binary market basket data. *Journal of Retailing and Consumer Services*, 10(3):123–133, 2003.

B. Mobasher, H. Dai, T. Luo, and M. Nakagawa. Effective personalization based on association rule discovery from web usage data. In *Proceedings of the ACM Workshop on Web Information and Data Management (WIDM01), Atlanta, Georgia*, 2001.

R. Pan, Y. Zhou, B. Cao, N. N. Liu, R. Lukose, M. Scholz, and Q. Yang. One-class collaborative filtering. In *IEEE International Conference on Data Mining*, pages 502–511, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994. ISBN 0-89791-689-1. doi: http://doi.acm.org/10.1145/192844.192905.

G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Analysis of recommendation algorithms for e-commerce. In *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*, pages 158–167. ACM, 2000. ISBN 1-58113-272-7.

B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001. ISBN 1-58113-348-0.

J. B. Schafer, J. A. Konstan, and J. Riedl. E-commerce recommendation applications. *Data Mining and Knowledge Discovery*, 5(1/2):115–153, 2001.

U. Shardanand and P. Maes. Social information filtering: Algorithms for automating 'word of mouth'. In *Conference proceedings on Human factors in computing systems (CHI'95)*, pages 210–217, Denver, CO, May 1995. ACM Press/Addison-Wesley Publishing Co.

C. van Rijsbergen. *Information retrieval*. Butterworth, London, 1979.

S. M. Weiss and N. Indurkhya. Lightweight collaborative filtering method for binary-encoded data. In *Principles of Data Mining and Knowledge Discovery*, volume 2168/2001 of *Lecture Notes in Computer Science*, pages 484–491. Springer, 2001.

M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, May/June 2000.