LEIBNIZ UNIVERSITÄT HANNOVER

FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK
INSTITUT FÜR VERTEILTE SYSTEME

Multi-purpose Library of Recommender System Algorithms for the Item Prediction Task

Bachelor Thesis

eingereicht von

JULIUS KOLBE

am 11. Juni 2013

Erstprüfer	:	Prof. Dr. techn. Wolfgang Nejdl
Zweitprüfer	:	Jun.-Prof. Dr. rer. nat. Robert Jäschke
Betreuer	:	Ernesto Diaz-Aviles, M. Sc.

Julius Kolbe: *Multi-purpose Library of Recommender System Algorithms for the Item Prediction Task,* Bachelor Thesis, © 11. Juni 2013

EHRENWÖRTLICHE ERKLÄRUNG

Hiermit versichere ich, die vorliegende Bachelor Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die wörtlich oder inhaltlich aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Hannover, den 11. Juni 2013

Julius Kolbe

ABSTRACT

In the context of this thesis, we wrote a library called recsyslab [12] in the programming language Python [11]. The goal of recsyslab is to provide easy access to current recommendation algorithms by being easy to use and having easy readable source code. The development of recsyslab is the major contribution of this thesis. This document is intended to document its design and facilitate its use. The user guide in chapter 7 shows that we have been able to make the library easy to use and the test results in chapter 5 show that it also performs well.

ZUSAMMENFASSUNG

Im Rahmen dieser Bachelorarbeit haben wir eine Bibliothek namens recsyslab [12] in der Programmiersprache Python [11] geschrieben. Das Ziel von recsyslab ist es, einen möglichst einfachen Zugang zu Empfehlungsalgorithmen zu gewähren, indem es einfach zu benutzen ist und der Quellcode gut zu lesen ist. Die Entwicklung von recsyslab ist der Hauptteil der Bachelorarbeit. Dieses Dokument soll ihre Struktur dokumentieren und ihren Gebrauch vereinfachen. Die Bedienungsanleitung in Kapitel 7 zeigt, dass es uns gelungen ist die Bibliothek einfach bedienbar zu gestalten und die Testergebnisse in Kapitel 5 zeigen, dass sie gut funktioniert.

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

The library together with this document shall provide a "cookbook" for recommender systems. Recommender systems are widely used in ecommerce and are an active field of research so an easy accessible library will be useful in the future. With the simple syntax and the interactivity of Python recsyslab is aimed at beginners to simply experiment with different algorithms and comprehend them. Together with this document everybody should be able to get started to try out different recommender algorithms without needing any previous knowledge or long configuration.

1.2 TASK (WHAT A RECOMMENDER SYSTEM DOES)

A Recommender System works in a scenario with users, items and interactions between these two. Such a scenario could be an online shop, where the interactions are purchases of items by users or a video platform, where the users interact with items (videos) by watching them e.g. youtube.com [15]. Based on the past interactions of the users a Recommender System presents them with a ranked list of items, which meets their interest.

Two common scenarios are:

- Rating Prediction: When the feedback is provided explicitly like ratings the scenario is called rating prediction. The rating prediction task got very popular when netflix announced a prize [9] in September 2009. The algorithm which performs best in predicting ratings would be granted the prize money.

- Item Prediction: When the interactions are implicit like purchases or clicks, then the scenario is called item prediction. But in this work the focus lies on implicit feedback or item prediction. However ratings can also be interpreted as the strength of implicit feedback. For example how often a user purchased an item. Some algorithms implemented in this library can use this information but none will explicitly predict ratings like it is usual in rating prediction scenarios because it is normally not useful to predict explicit absolute ratings when you want to propose items a user could find interesting. In this case it is sufficient to only have a ranked list of items according to the preferences of the users. Most of the recommender algorithms in recsyslab compute a score which is used to rank the items relative to each other. But this score is not an absolute rating.

1.3 CONTRIBUTIONS

The contributions of this bachelor thesis are:

1. recsyslab with state of the art algorithms
2. supporting infrastructure for testing recommender algorithms
3. easy to use library
4. easy readable source code
5. extensive user manual
6. test results of the algorithms computed with recsyslab
7. explanations of the technical background

1.4 STRUCTURE OF THIS DOCUMENT

In the next chapter is an overview over the research area of recommender systems together with explanations of different evaluation metrics and the Leave-one-out Protocol used to evaluate recommender algorithms. After that we provide short descriptions over already existing projects providing something similar to recsyslab. In chapter 4 we present the implemented recommender algorithms together with the core of their implementation. This is followed by a chapter with results and explanations about how the results have been obtained. After that we will show the inner structure of recsyslab to make it easy for developer to extend or change the library. In chapter 7 is an extensive guide for recsyslab explaining how the library is used. In the last chapter are the conclusions and an outlook to future work.

2 BACKGROUND

In this chapter we will provide a short overview over the area of recommender system and how to test them.

2.1 COLLABORATIVE FILTERING

There are two approaches for recommender systems:

- Collaborative filtering
- Content-based filtering

Collaborative filtering is in general the process of finding informations or patterns in large datasets. In the context of recommender systems it is refered to in a narrower sense where the task is to predict user preferences. By assuming that users who have the same opinion on some items will also agree on other it is possible to do predictions without using additional informations like the age of the users [2]. Content-based filtering uses informations about users and items like, for example, the age of the users or the names of the items [3]. This work is only about collaborative filtering so when we are speaking about recommender systems we mean recommender systems using the collaborative filtering approach.

2.2 MATRIX FACTORIZATION

An important group of recommender algorithms use matrix factorization. Matrix factorization splits a large matrix into a product of matrices [7]. In recommender systems the matrix representing the dataset M is split into two matrices W and H so that $M = W \times H$. The matrices W and H are supposed to represent abstract features of each item and user. For recommendation the dot product of the feature vector of an user and an item gives a score whith which we can sort the items and recommend the best suitable ones.

2.3 EVALUATION METHODS

To evaluate a recommender algorithm we have to split up the database into one for training and one for evaluation. There are different methods to split the database but in the library only one is implemented which is the Leave-one-out protocol [5]. You can use the Leave-one-out protocol with many different metrics which are also explained here.

2.3.1 *Leave-one-out Protocol*

The Leave-one-out protocol works as follows:

1. Randomly choose one interaction by user. These are the hidden interactions.
2. Put the hidden interactions into the test set.
3. Put all other interactions into the training set.
4. Let the recommender recommend N items for every user by ranking all items, except the ones the user already interacted with, and recommending the first N items .
5. If the hidden interaction of the user is in the recommendations for this user, the recommender got a hit.
6. Count the hits and record the position of the interaction in the recommendations.
7. Use these informations to compute the various metrics.

Because of the split we can test the recommender on every user. This would not be possible if we would for example split by randomly choosing 1% of the interactions. Then it would be likely that there are some users without a hidden item. And without a hidden item it would be impossible to measure the quality of the recommendations for this user.

The recommender algorithm recommends N items. This N has to be chosen according to the deployment scenario. It depends for example on the layout of the site how many items can get recommended at once. According to a chosen N and with the metrics of recsyslab the right algorithm can be found.

Except AUC and MRHR the metrics are not taking the position of the items in the recommendation list into account. What is more important also depends on the use case. When all recommended items are presented equally well to the user the position of the items in the recommendation is not important, only if or if not the right items are recommended. But in contrast to this when the user is shown a long list of items and it is likely that the user will not scroll down until the end of the list you would rather care about the position of the items in the list of recommendations.

2.3.2 *Evaluation metrics*

These are a selection of different metrics to rate the recommendations. By default the evaluations are executed with only one hidden item but generally the metrics should also work with more than just one.

For the notation: U is the set of users, H is the set of hidden items and H_u is the set of hidden items for user u. T is the set used for training. $TopN_u$ is the set of top N recommendations for user u so the number of items the recommender is allowed to recommend is N. For the metrics where the order in which the items are recommended count TopN_u is a list sorted by score in decreasing order. To get an implicit score of each item the recommender recommends all items.

2.3.2.1 *Hitrate/Recall@N*

This metrics lets the recommender recommend N items. If the hidden item is under the N recommended items, the recommender got a hit [22, 27]. So the Recall@N is the fraction of users who get recommended a relevant item when the recommender can recommend N items. So the hitrate is

$$\text{Recall@N} = \frac{\sum_{u \in U} H_u \cap \text{topN}_u}{|H|} \tag{2.1}$$

This metric is very intuitive you can for example imagine that you show the user 10 items then Recall@10 would be the chance of showing the user an item he will interact with. But this metric does not take the number of recommended items into account.

2.3.2.2 *Precision*

The precision [27] is

$$\text{Precision} = \frac{\sum_{u \in U} H_u \cap \text{topN}_u}{N \times |U|} \tag{2.2}$$

As you can clearly see this metric is taken the number of recommended items into account. Which will probably lead to worse results as the number of recommended items increases.

2.3.2.3 *F1*

The F1 metric [27] tries to balance hitrate and precision by taking both into account.

$$\text{F1} = \frac{2 \times \text{Recall@N} \times \text{Precision}}{\text{Recall@N} + \text{Precision}} \tag{2.3}$$

2.3.2.4 *Mean Reciprocal Hitrate*

The mean reciprocal hitrate or more general mean reciprocal rank [25] counts the hits but punishes them the more the lower they appear in the list of recommendations. So if the hidden item appears first in the list of recommendations the hit counts as one, but when it is in the second position the hit already counts only as one half and so on.

$$\text{MRHR} = \frac{1}{|U|} \sum_{u \in U} \frac{1}{\text{pos}(\text{topN}_u, H_u)} \tag{2.4}$$

Where N is the number of items in the dataset and $pos(topN_u, H_u)$ is the position of the hidden item in the recommendation.

2.3.2.5 *Area under the ROC (AUC)*

AUC [26] counts the number of items the recommender rates higher than the hidden item, normalize it by the number of items the recommender can rate higher. Sum this up for every user and again normalize by the number of users.

To get an implicit score of each item the recommender recommends all items in a list sorted by score in decreasing order. This is in fact the same as for the other metrics only that the recommender can recommend as many items as possible.

$$\mathrm{AUC} = \frac{1}{|U|} \sum_{u \in U} \frac{1}{|E(u)|} \sum_{(i,j) \in E(u)} \delta(x_{ui} > x_{uj}) \tag{2.5}$$

Where x_{ui} is the predicted score of the interaction between User u and item i. δ is defined as follows

$$\delta(x) = \begin{cases} 1, & \text{if x is true} \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

And $E(u)$ is

$$E(u) = \{(i,j) | (u,i) \in H \wedge (u,j) \notin (H \cup T)\} \tag{2.7}$$

2.4 DATASETS FOR TESTING

In the WWW there are several anonymized datasets available to try out recommender systems and to evaluate their performance. Following we will introduce three of them.

2.4.1 *MovieLens*

MovieLens [8] is a database provided by GroupLens, a research lab at the University of Minnesota. One of their research areas is recommender systems and they built an application where users rate movies and then get recommendations for movies the could like. The MovieLens dataset is the ratings gathered by this application. For this work we will interpret the rating as intensity of interaction between users and items for example the number of times the user saw this movie.

The dataset is available in three different sizes:

- 100,000 interactions
- 1 million interactions
- 10 million interactions

For the experiments the smallest dataset is totally sufficient, with the larger datasets the computation time gets too long for just trying something out.

2.4.2 *Million Song Dataset*

The million song dataset [16] is a large database of features and media data of a million songs. For a challenge they also provided the listening history of over 1 million user. To present I will use a subset of this dataset to keep the computing time required reasonable low so it is easier for others to retrace the results.

2.4.3 *476 Million Twitter Tweets Dataset*

The Stanford Network Analysis Project provided a twitter dataset with about 467 million tweets from 17.000 users [13]. Unfortunately the dataset is no more available. [further explanation or deletion] To convert the tweets two user item interactions I will interpret the hashtags[explanation necessary?] as items. So tweets of a user with a hashtag is a interaction between the user and the hashtag.

3 RELATED WORK

There is a wide range of projects providing implemantions for recommender system. Some of them are described in this chapter to give a quick overview and comparison.

3.1 MYMEDIALITE

MyMediaLite [20] is an open source project developed at the University of Hildesheim and provides several algorithm for rating prediction and item prediction. It is written in C# and is used with a command line interface. It also provides a graphical interface to demonstrate recommender algorithms

3.2 PREA (PERSONALIZED RECOMMENDATION ALGORITHMS TOOLKIT)

PREA [23] is an open source project written in Java. It provides a wide range of recommender algorithms and evaluation metrics to test them. It is maintained by the Georgia Institute of Technology.

3.3 APACHE MAHOUT

Mahout [6] is an open source library in java. It is implemented on top of Apache Hadoop, so it uses the map/reduce paradigm. This means it can run on different independent computers.

3.4 DUINE FRAMEWORK

The Duine Framework [4] is an open source project written in java by the Telematica Instituut/Novay. The recommender of the Duine Framework combines multiple prediction techniques to exploit the strengths of the different techniques and to avoid their weaknesses.

3.5 COFI

Cofi [1] provides an algorithm for the rating prediction task called Maximum Margin Matrix Factorization. It is open source and written in C++.

3.6 LENSKIT

Lenskit [19] is a toolkit which provides several recommender algorithms and an infrastructure to evaluate them. It is an open source project by the University of Minnesota

3.7 COMPARISON

Following a table comparing the algorithms implemented by the different frameworks.

Category	Feature	PREA	Mahout	Duine	Cofi	MyMedia	recsyslab
Baselines	Constant	O			O	O	O
	User/Item Average	O	O	O	O	O	
	Random	O				O	O
Memory-based CF	User-based CF (Su and Khoshgoftaar, 2009)	O	O	O	O	O	O
	Item-based CF (Sarwar et al., 2001)	O	O	O	O	O	O
	Default Vote, Inf-User-Frex (Beese et al., 1998)	O	O				
	Slope-One (Lemire and Maclachlan, 2005)	O	O			O	O
Matrix Factorization	SVD (Paterek, 2007)	O	O		O	O	O
	NMF(Lee and Seung, 2001)	O					
	PMF(Salakhutdinov and Mnih, 2008b	O					
	Bayesian PMF (Salakhutdinov and Mnih, 2008b)	O					O
	Non-linear PMF (Lawrence and Urtasun, 2009)	O					
	RankMFX (???)						O
Other methods	Fast NPCA (Yu et al., 2009	O					
	Rank-based CF (Sun et al., 2011, 2012)	O					
Evaluation Metric	(N)MAE	O	O	O	O	O	
	RMSE	O	O		O	O	
	HLU/NDCG	O				O	
	Kendall's Tau, Spearman	O					
	Precision/Recall/F1		O			O	O
	ARHR/MRHR						O
Miscelaneous	Sparse Vector/Matrix	O	O	O	O	O	O
	Wrapper for other languages	O			O	O	?
	Item Recommender for Positive-only Data					O	O
	Release Year	2011	2005	2009	2004	2009	2013
	Language	Java	Java	Java	Java	C#	Python
	License	GPL	LGPL	LGPL	GPL	GPL	???

4 RECOMMENDATION ALGORITHMS

In this chapter we will provide an overview on how the algorithms we implemented work. To illustrate the simplicity of the code we will also show the core part of the source code of the respective recommender algorithm. For further explanations please refer to the cited papers.

4.1 NON-PERSONALIZED ALGORITHMS

In this chapter we will describe two very simple and basic recommendation algorithms we implemented for comparison with the more sophisticated algorithms.

4.1.1 *Constant*

The constant recommender algorithm counts the number of interactions for each item and sorts this in decreasing order of interactions. Then it recommends the top items of this list. So it recommends the items which are the most popular over all users and does not do any personalizations. The intuition here is that the most popular items will be interesting for everyone. However the results show that algorithms which personalize the recommendation based on the interaction history of the users perform much better. In pseudocode this algorithm will look like this:

```
countDict = dict initialized with all ItemIDs as keys and 0 as values

for interaction in dataset:
    countDict(ItemID(interaction))++

ranking = list(ItemIDs in decreasing order of their values)

return first N items of ranking
```

And the implementation:

```
def __init__(self, dbdict):
    self.dictionary = {}
    self.sortedList = []
    for data in dbdict.iteritems():
        for item, rating in iter(data[1]):
            if item in self.dictionary:
                self.dictionary[item] += rating
            else:
                self.dictionary[item] = rating

    self.sortedList = helper.sortList(self.dictionary.iteritems())
```

```
def getRec(self, user, n):
    return self.sortedList[:n]
```

4.1.2 *Random*

The random recommender algorithm chooses items to recommend randomly. Even though this algorithm will recommend different items for different users it is not personalized because it the recommendations are independent of the previous interactions of the user. In pseudocode this will be:

```
return N randomly chosen items
```

And in Python:

```
def __init__(self, dbdict, seed):
    self.maxIid = 0
    self.seed = seed
    for data in dbdict.iteritems():
        for itemRating in iter(data[1]):
            item = itemRating[0]
            if item > self.maxIid:
                self.maxIid = item
    self.maxIid += 1

def getRec(self, user, n):
    random.seed(self.seed)
    if self.maxIid < n or n == -1:
        l = range(self.maxIid)
        random.shuffle(l)
        return l
    return list(random.sample(range(self.maxIid), n))
```

4.2 K-NEAREST-NEIGHBOR

This class of recommendation algorithms works by searching neighbors of either items or users based on a similarity function which is the cosine in this library. The similarity function is interchangeable for example in [22] two similarity functions are compared. The cosine similarity performs best so in recsyslab only the cosine similarity is implemented. For two vectors $\vec{v}, \vec{u}$ The cosine similarity is defined as follows:

$$\cos(\vec{v}, \vec{u}) = \frac{\vec{v} \cdot \vec{u}}{||\vec{v}||_2 ||\vec{u}||_2} \tag{4.1}$$

With the similarity it is possible to rank items. In the Item Based algorithm for each item we compute the sum of similarities with the items the user already bought and rank accordingly. In the User Based algorithm for every item we sum up the similarities of the user who bought this item and rank according to this score.

4.2.1 *Item Based*

For this algorithm the database has to be represented as a matrix where the rows correspond to the users and the columns to the items. Then the entry (i,j) represents the number of transactions which happened between the ith user and the jth item.

The algorithm interprets the columns of the matrix i.e. the items as vectors and computes there similarities by computing their cosine. To build the model the algorithm computes the n most similar items of each item. In pseudocode:

```
for every item i
    for every item j
        sim[i,j] = similarity between i and j

for every item i
    for every item j
        if sim[i,j] not one of the n largest in sim[i]
            sim[i,j] = 0
```

In Python it looks like this:

```
def __init__(self, userItemMatrix, n):
    self.itemUserMatrix = userItemMatrix.transpose()
    self.sim = computeCosSim(self.sim, self.itemUserMatrix)
    self.userItemMatrix = userItemMatrix

    order = self.sim.argsort(1)

    # for each row in sim:
    # Set all entries to 0 except the n highest
    for j in xrange(0, self.sim.shape[1]):
        for i in xrange(0, self.sim.shape[1] - n):
            self.sim[j, order[j, i]] = 0
```

`computeCosSim` looks like this:

```
def computeCosSim(sim, matrix):
    for i in xrange(1, sim.shape[1]):

        for j in xrange(0, i):
            sim[i, j] = sim[j, i] = cos(
                matrix[i], matrix[j])
    return sim
```

This function is also used in the user based k-Nearest-Neighbor.

To compute recommendations for user U the algorithm computes the union of the n most similar items of each item U interacted with. From this set the items U already interacted with are removed. For each item remaining in this set we compute the sum of its similarities to the items U interacted with. Finally these items are sorted in decreasing order of this sum of similarities and the first n items will be recommended [22]. The pseudocode is

```
for every item i user u bought
    itemSimVector = itemSimVector + vector of i

for every item i user u bought
    itemSimVector[i] = 0

return the N items with the highest value in itemSimVector
```

The Python equivalent:

```
def getRec(self, u, n):
    # x are the similarities of each item to the items u bought
    # w.r.t. only the highest n similarities are saved.
    x = self.userItemMatrix[u] * self.sim

    # Throw out items the user already purchased
    for i in xrange(0, self.sim.shape[0]):
        if self.userItemMatrix[u, i] != 0:
            x[0, i] = 0

    order = x.argsort()
    l = []
    for i in xrange(1, n + 1):
        l.append(order[0, -i])
    return l
```

4.2.2 *User Based*

The user based k-Nearest-Neighbor [14] is very similar to the item based. But instead of interpreting the columns as vectors we interpret the lines or users of the matrix as vectors and compute their similarities to other users. Pseudocode:

```
for every user u
    for every user o
        sim[u,o] = similarity between u and o

for every user u
    for every user o
        if sim[u,o] not one of the n largest in sim[u]]
            sim[u,o] = 0
```

Python:

```
def __init__(self, userItemMatrix, n):
    self.userItemMatrix = userItemMatrix
    self.sim = np.zeros((userItemMatrix.shape[0], userItemMatrix.shape
        [0]))
    self.sim = computeCosSim(self.sim, self.userItemMatrix)

    order = self.sim.argsort(1)

    # for each row in sim:
```

```
    # Set all entries to 0 except the n highest
    for j in xrange(0, self.sim.shape[1]):
        for i in xrange(0, self.sim.shape[1] - n):
            self.sim[j, order[j, i]] = 0
```

Then for each item i we sum up the similarities between U and the users who interacted with i. Again we remove all items U already interacted with, sort in decreasing order fo the sum and recommend the first n items. Pseudocode:

```
for every item i
    for every user o
        score(i) = score(i) + sim[U,o]

for every item i user u bought
    score(i) = 0

return the N items with the highest value in score
```

Python:

```
def getRec(self, u, n):
    # x is the weighted sum of the items
    # weighted with the similarity between u and the
    # other users.
    x = self.sim[u] * self.userItemMatrix

    for i in xrange(0, self.sim.shape[0]):
        if self.userItemMatrix[u, i] != 0:
            x[0, i] = 0

    order = x.argsort()
    l = []
    for i in xrange(1, n + 1):
        l.append(order[0, -i])

    return l
```

4.3 MATRIX FACTORIZATION

As mentioned in 2.2 the matrix factorization techniques generate two matrices W and H so that $M = W \times H$. Each of the implemented algorithms train these two matrices with stochastic gradient descent. In each iteration the model is trained with

- a randomly chosen user U
- a randomly chosen item I the user U interacted with, called the positive item and
- a randomly chosen item J the user U did not already interacted with, called the negative item.

The features of U, I and J are trained using the derivative of a loss function.

BPRMF and RankMFX are sharing the code in the module `recommender.mf`. The only difference is the loss function. In pseudocode the model update happening in each iteration looks like this:

```
U = randomly chosen user
I = randomly chosen item U interacted with
J = randomly chosen item U did not interact with

X=H[i] - H[j]
wx = dot product of W[u] and X
dloss = (derivative of the loss function of wx and 1) * learningRate

W[u] += dloss * (H[i] - H[j]) # These three lines
H[i] += dloss * W[u]          # have to be
H[j] += dloss * -W[u]         # executed at once
```

The Python code doing the same looks like this:

```
u = random.choice(R.keys())

userItems = [x[0] for x in R[u]]
# the positive example
i = userItems[np.random.random_integers(0, len(userItems) - 1)]
# the negative example
j = np.random.random_integers(0, m_items)
# if  j is also relevant for u we continue
# we need to see a negative example to contrast the positive one
while j in userItems:
    j = np.random.random_integers(0, m_items)

X = H[i] - H[j]
wx = np.dot(W[u], X)
dloss = dlossF(wx, y)

# temp
wu = W[u]
hi = H[i]
hj = H[j]

if dloss != 0.0:
    # Updates
    eta_dloss = learningRate * dloss
    W[u] += eta_dloss * (hi - hj)
    H[i] += eta_dloss * wu
    H[j] += eta_dloss * (-wu)

    W[u] *= scaling_factorU
    H[i] *= scaling_factorI
    H[j] *= scaling_factorJ
```

For the recommendations we rank the items with a score. The score of an item for a user is the dot product of the feature vector of the user and the feature vector of the item. All matrix factorization algorithms use this. In pseudocode:

```
for every item i
    scoreList(i) = W[u] * H[i]

sortByScore(score)
return first N of score
```

And in Python:

```
def getRec(self, u, n):
    scoredict = {}
    for i in range(0, self.H.shape[0]):
        if not i in [x[0] for x in self.R[u]]:
            scoredict[i] = np.dot(self.W[u], self.H[i])

    if n == -1:
        n = len(scoredict)
    import util.helper
    return util.helper.sortList(scoredict.iteritems())[:n]
```

4.3.1 *BPRMF*

BPMRF uses the logloss to train the model. The logloss is defined as

$$\text{logLoss}(a, y) = \log(1 + \exp(-ay)) \tag{4.2}$$

And the derivative of the log loss is

$$\frac{\partial}{\partial y}(\log(1 + \exp(-ay))) = -\frac{a}{\exp(ay) + 1} \tag{4.3}$$

For further informations please refer to [26]

4.3.2 *RankMFX*

RankMFX uses the hingeLoss. It is defined as

$$\text{hingeLoss}(a, y) = \max(0, 1 - ay) \tag{4.4}$$

And its derivative

$$\frac{\partial}{\partial y}(\max(0, 1 - ay)) = \begin{cases} -y & ay < 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

See also [18].

4.3.3 *Ranking SVD (Sparse SVD)*

Ranking SVD uses the quadratic loss and the difference between the predicted score of the positive item and the negative minus the actual score of the positive item [21]. Ranking SVD uses another code than BPRMF and RankMFX but the iterations work the same. It also randomly chooses a user and a positive and a negative item in each iteration of the stochastic gradient descent. But the model updates are different. In pseudocode:

```
U = randomly chosen user
I = randomly chosen item U interacted with
J = randomly chosen item U did not interact with

ruipred = W[U] * H[I] # Prediction for item I
rui0pred = W[U] * H[J] # Prediction for item J
rui = actual value of the interaction between U and I
ruj = 0

dloss = (ruipred - rujpred) - (rui - ruj)
W[U] = W[U] - learningRate * dloss * (H[I] - H[J])
H[I] = H[I] - learningRate * dloss * W[U]
H[J] = H[J] - learningRate * dloss * -W[U]
```

And in Python this part looks like this:

```
# Choose an user randomly
u = random.choice(R.keys())
# Choose an item the user interacted with
userItems = [x[0] for x in R[u]]
i = random.choice(userItems)
# Choose an item, the user didn’t interacted with
i0 = np.random.random_integers(0, m_items - 1)
while i0 in userItems:
    i0 = np.random.random_integers(0, m_items - 1)

# Prediction for the first item
ruipred = userFeatures[u].dot(itemFeatures[i])
# Prediction for the second item
rui0pred = userFeatures[u].dot(itemFeatures[i0])
rui0 = 0
for r in R[u]:
    if r[0] == i:
        rui = r[1]

dloss = (ruipred - rui0pred) - (rui - rui0)
c = userFeatures[u]
userFeatures -= learningRate * dloss * (
    itemFeatures[i] - itemFeatures[i0])
itemFeatures[i] -= learningRate * dloss * c
itemFeatures[i0] -= learningRate * dloss * (-c)
```

4.4 OTHER

In this chapter we will describe one last algorithm which does not fit into the above categories.

4.4.1 *Slope One*

The Slope One recommendation algorithm computes the differences of interaction intensities between items and uses these differences to predict indirectly the interaction intensity between users and items without any interaction [24]. The model building in pseudocode:

```
for all interaction pairs i, j where i != j
    diffs[i][j] = i.rating - j.rating
    count[i][j]++
```

And in Python:

```
def __init__(self, R):
    self.diffs = {}
    self.R = R

    for u in R.keys():
        for i, r in R[u]:
            if not i in self.diffs:
                self.diffs[i] = {}

            for i1, r1 in R[u]:
                if i == i1:
                    continue
                if not i1 in self.diffs[i]:
                    self.diffs[i][i1] = [0.0, 0.0]

                self.diffs[i][i1][0] += r - r1
                self.diffs[i][i1][1] += 1
```

For the recommendations for every item we sum up its differences with the items the user rated together with the rating, the user gave these other items. Additionaly this is multiplied by the number of times the difference occured. This is then normalized by the sum of the occurrences of the differences. Following is pseudocode to clarify this.

```
for every item i
    ratingSum = 0
    count = 0
    for every item j user u interacted with
        ratingSum = ratingSum + ((rating u gave j)
                          + diffs[i][j])
                          * count[i][j]
        count = count + count[i][j]

    ratingSum /= count
    listToRecommend.add( (i, ratingSum))
```

```
sortAfterRatingSum(listToRecommend)
return first N of listToRecommend
```

And in python:

```
def getRec(self, u, n):
    maxIid = max(self.diffs.keys())
    userItems = [x[0] for x in self.R[u]]
    predictionList = []

    for i in xrange(0, maxIid + 1):
        if i in userItems:
            continue

        ratingSum = 0
        count = 0
        for i1, r in self.R[u]:
            if i in self.diffs:
                if i1 in self.diffs[i]:
                    ratingSum += (r + self.diffs[i][i1][0]
                                ) * self.diffs[i][i1][1]
                    count += self.diffs[i][i1][1]

        if count == 0:
            continue
        ratingSum /= count
        predictionList.append((i, ratingSum))

    import util.helper
    sortedList = util.helper.sortList(predictionList)

    return sortedList[:n]
```

5 EXPERIMENTS

In this chapter we will show test results of every metric with every recommender algorithm and how they were computed.

5.1 EXECUTION

The results were computed using the recommender algorithms and the test metrics implemented in recsyslab. It was done like it is shown in the user manual in 7. with exact the same parameters. The only difference is that for simplification the different getRec methods and metric methods were each in a list and the algorithm is iterating trough both in two nested for loops to guarantee that every algorithm is tested with every metric. The code which generated the table in 5.2 is in the experiments.py file in the bin directory of recsyslab. So check it out for further implementation details.

5.2 RESULTS

Here are the results of every test metric in recsyslab with every recommender algorithm in recsyslab.

recommender	hitrate	precision	f1	mrhr	auc
constant	0.0731	0.0073	0.1448	0.0207	0.8263
randomRec	0.0053	0.0005	0.0104	0.0011	0.4790
itemKnn	0.2576	0.0257	0.5099	0.1151	0.8687
userKnn	0.2619	0.0261	0.5183	0.1203	0.9279
BPRMF	0.2492	0.0249	0.4931	0.0996	0.9294
RankMFX	0.1792	0.0179	0.3546	0.0686	0.8295
Ranking SVD	0.1198	0.0119	0.2371	0.0339	0.7725
slopeone	0.0858	0.0085	0.1699	0.0350	0.4699

5.3 COMPARISON

To show the performance of recsyslab we compare our test results with results from this paper: [17]

	hitrate		mrhr	
	recsyslab	[17]	recsyslab	[17]
constant	0.0731	0.131	0.0207	0.046
itemKnn	0.2576	0.271	0.1151	0.119
userKnn	0.2619	0.281	0.1203	0.128

It can be seen that recsyslab performs almost as good as the reference. The small advantage probably comes from normalization we did not use when we were computing these results.

6 DESIGN AND IMPLEMENTATION

In this chapter we would like to give a quick overview over the library by explaining the overall structure of the library and how to extend it.

Dependencies

To run recsyslab you need a Python 2.7.x interpreter [11] and NumPy [10], a package for scientific computing with Python.

Input Dataset Format

Right now only one type of datasets is supported. The dataset has to be a textfile where each line is of this form:

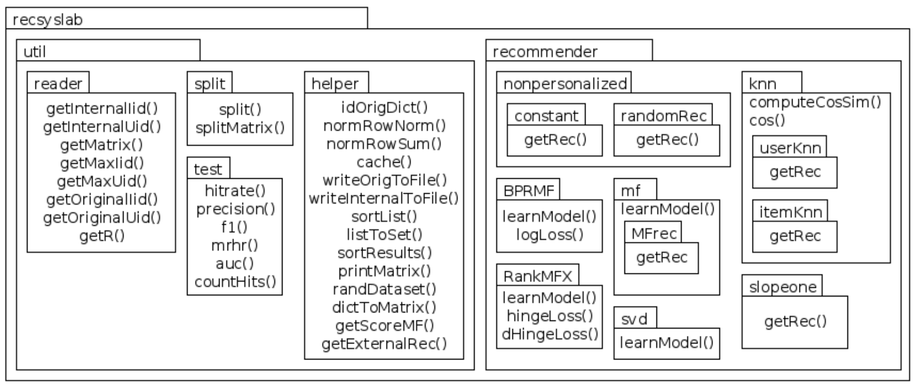
```
UserID<separator>ItemID<separator>NumberOfInteractions
```

<separator> is an arbitrary string but it has to be the same throughout the whole dataset. NumberOfInteractions is optional and can be omitted. In this case one will be assumed. Everything coming after NumberOfInteractions<separator> will be ignored. Please note that when you are omitting NumberOfInteractions but have something else after the ItemId, this will be recognized as NumberOfInteractions.

We recommend to use the MovieLens database2.4.1 with 100,000 ratings. It is easy to get, does not need any modifications to work with our library and has a reasonable size. Also we will use this dataset in the examples of the user manual in chapter 7.

6.1 GENERAL STRUCTURE

Here is a figure illustrating the structure of recsyslab:



The util Package

The util package contains several modules for the things happening around the recommender algorithms and the methods for the test metrics. The modules in the util package are:

READER manages the data

SPLIT splits up the dataset following the leave-one-out protocol, see also chapter 2

HELPER contains several helper functions

TEST contains the methods for computing the test metrics

The reader module is a very central part in recsyslab. The class reader in the module reader takes care of reading the dataset and supplies the other parts of recsyslab with every kind of data they need. While the reader reads the dataset it maps the UserIDs and ItemIDs from the original IDs in the dataset to internal IDs starting at zero and counting up to guarantee that the IDs are consecutive. When the IDs are consecutive it is for example possible to use them as indices in a matrix.
Furthermore the reader constructs a dict and a matrix out of the data using the internal IDs. The dict has the internal UserIDs as keys and the values are tuples representing the interactions of the corresponding user. The first value of these tuples is the internal ItemID of the interaction, the second is the intensity or quantity of the interaction. You get the dict by calling `getR()` of the reader object.
The matrix offered by a reader has as much lines as the dataset has users and as much columns as items are present in the dataset. The entry at the ith line and jth column is then the intensity of the interaction between user i and item j. A zero means that there has not been any interaction yet.
The k-Nearest-Neighbor algorithms need a matrix like it is provided by a reader. All other recommendation algorithms need a dict of the form described above.

The recommender Package

In the recommender package are all recommender algorithms. They are separated into the following modules:

NONPERSONALIZED simple, nonpersonalized algorithms

KNN k-Nearest-Neighbors algorithms

MF Pairwise matrix factorization algorithms

BPRMF Bayesian Personalized Ranking Matrix Factorization algorithm

RANKMFX RankMFX algorithm

SLOPEONE Slope One algorithm

SVD Ranking SVD (Sparse SVD)

For descriptions of the algorithms please refer to chapter 4. To implement new recommender algorithms which integrate nicely into the infrastructure the only thing you have to take care of is to provide a getRec function which has an internal UserID as first parameter and the number of items to recommend as the second parameter. Also it should return a list of internal ItemIDs of the length which got specified in the second parameter.

7 USER MANUAL

In this chapter we will provide a step by step user manual for recsyslab.

To run recsyslab you need to install Python [11] and NumPy [10]. Please refer to their sites for informations on how to install them.

First, we will explain how to load a dataset, second we will explain how to use the different recommendation algorithms and how to test them with the provided evaluation metrics.

For informations about the recommendation algorithms, please refer to chapter 4. For informations about the test metrics refer to chapter 2.

The following python code is also in the file manual.py in the bin directory of the library so you do not have to copy the code out of this pdf document. But if you are working through these examples please note that you perhaps miss the import of a module when you skip an example because We will not write each needed import in each new example.

Also for help you can use the inline documentation available as docstrings. You can display them with the command line utility[1] pydoc. So for example when you are in the bin directory of the library call

```
$ pydoc __init__
$ pydoc util
$ pydoc recommender.BPRMF
```

Each of these commands will show the documentation of the specified module.

7.1 LOAD THE DATASET

In chapter 6 we explained how the dataset has to look like. For trying things out the MovieLens dataset [8] would work just fine. When you have a dataset, place it into the bin directory of recsyslab and start the python interpreter from the command line with

```
$ python
```

Now import the util.reader module and initialize a new reader object with [2]

```
>>> import util.reader
>>> r=util.reader.stringSepReader("u.data","\t")
Start reading the database.
10000 Lines read.
20000 Lines read.
30000 Lines read.
40000 Lines read.
50000 Lines read.
```

1 We will use $ to indicate a bash prompt.
2 We will use >>> to indicate the python prompt.

```
60000 Lines read.
70000 Lines read.
80000 Lines read.
90000 Lines read.
100000 Lines read.
```

Note that it outputs the progress it has already made. The first parameter of the constructor is the name of the file containing the dataset, the second the string which is separating the values, here it is a tab. When you are using another dataset you probably have to change the filename and perhaps also the separating string.

Mapping

The constructor creates a mapping from the original IDs to internal IDs both for the users and the items to make sure that the IDs are consecutive. So to get the items a user interacted with we have to first find out the internal UserID.

```
internalID=r.getInternalUid("196")
>>> r.getR()[internalID]
set([(521, 1), (377, 4), (365, 3), (438, 5), (86, 4), (649, 4), (0, 3)
    , (522, 3), (423, 3), (389, 5), (751, 3), (656, 4), (947, 4), (432,
     2), (632, 2), (431, 5), (221, 5), (92, 4), (291, 3), (528, 4),
    (83, 4), (363, 3), (466, 4), (289, 5), (512, 5), (179, 3), (329, 4)
    , (672, 4), (834, 5), (665, 3), (321, 2), (487, 3), (380, 4),
    (1006, 4), (1045, 3), (491, 3), (302, 4), (550, 5), (10, 2)])
```

r.getR() returns a dict with internal UserIDs as keys and sets of (ItemID, NumberOfInteractions) tuples. Please note that the original ID is a string.

To evaluate the algorithms you have to split the datasets like described at 2.3.1. You do this by calling

```
>>> import util.split
>>> trainingDict, evaluationDict = util.split(r.getR(), 1234567890)
0 Users split.
100 Users split.
200 Users split.
300 Users split.
400 Users split.
500 Users split.
600 Users split.
700 Users split.
800 Users split.
900 Users split.
```

This will split a dict like r.getR() returns into a trainingDict where one transaction per user is missing and an evaluationDict with these missing transactions.

7.2 NON-PERSONALIZED ALGORITHMS

To make our first simple recommendations with the constant recommender we need to initialize an object of the constant class. As parameter the constructor needs a dict for training

```
import recommender.nonpersonalized
constant = recommender.nonpersonalized.constant(trainingDict)
constant.getRec(0, 10)
```

Every recommender has a getRec function with this signature. The first paramater is the internal UserID, the second is the number of items to be recommended. If n = -1 all items get recommendend. Also the IDs of the recommended items are internal ones. If you want to pass external UserIDs and get external ItemIDs back you do not have to map them all by yourself. Instead you can use a helper function called getExternalRec.

```
import util.helper
externalConstantgetRec =
    util.helper.getExternalRec(constant.getRec, r)
externalConstantgetRec("196", 10)
```

You pass the getRec function and the reader object and you get a new function which takes and returns only the original IDs.

Now we want to find out how good this algorithm is performing. Excpet AUC the functions for computing the metrics work by letting the recommender recommend for every user as much items as specified in the third parameter. With these recommendations and the hidden items the methods then calculate the performance according to the respective metric. The other parameters are a dict with the hidden items and the getRec function of the respective recommender algorithm.

First we will compute the hitrate

```
>>> import util.test
>>> util.test.hitrate(evaluationDict, constant.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 4.0
200 users tested
Hits so far: 14.0
300 users tested
Hits so far: 26.0
400 users tested
Hits so far: 31.0
500 users tested
Hits so far: 40.0
600 users tested
Hits so far: 49.0
700 users tested
Hits so far: 52.0
800 users tested
Hits so far: 61.0
```

```
900 users tested
Hits so far: 67.0
Number of hits: 69.0
Number of possible hits: 943.0
Hitrate: 0.07317073170731707
```

After some progress output it prints the number of hits and the number of possible hits which is in our case also the number of users. The hitrate is then its quotient.

Next we will calculate the precision

```
>>>util.test.precision(evaluationDict, constant.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 4.0
200 users tested
Hits so far: 14.0
300 users tested
Hits so far: 26.0
400 users tested
Hits so far: 31.0
500 users tested
Hits so far: 40.0
600 users tested
Hits so far: 49.0
700 users tested
Hits so far: 52.0
800 users tested
Hits so far: 61.0
900 users tested
Hits so far: 67.0
Number of hits: 69.0
Number of possible hits: 943.0
Precision: 6.9
```

The output looks similar to the one of hitrate and also means quite the same.

Very similar we can calculate the F1 metric

```
>>> util.test.f1(evaluationDict, constant.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 4.0
200 users tested
Hits so far: 14.0
300 users tested
Hits so far: 26.0
400 users tested
Hits so far: 31.0
500 users tested
Hits so far: 40.0
600 users tested
```

```
Hits so far: 49.0
700 users tested
Hits so far: 52.0
800 users tested
Hits so far: 61.0
900 users tested
Hits so far: 67.0
Number of hits: 69.0
Number of possible hits: 943.0
F1: 0.14480587618048268
```

Also the Mean reciprocal hitrate

```
>>> util.test.mrhr(evaluationDict, constant.getRec, 10)
0 users tested
Score so far: 0.0
100 users tested
Score so far: 0.675
200 users tested
Score so far: 2.9972222222222227
300 users tested
Score so far: 7.290079365079365
400 users tested
Score so far: 8.34404761904762
500 users tested
Score so far: 10.305158730158729
600 users tested
Score so far: 12.464682539682537
700 users tested
Score so far: 12.982539682539679
800 users tested
Score so far: 18.332539682539675
900 users tested
Score so far: 19.56825396825396
Score: 19.79325396825396
Number of possible hits: 943.0
Mean Reciprocal Hitrate: 0.02098966486559275
```

To compute the AUC the third parameters has to be our reader object because the function needs additional information to compute its metric. We do not need to specify the number of items to recommend because the AUC works by comparing all items so it needs the recommender to recommend all available items. The first two parameters do not change.

```
>>> util.test.auc(evaluationDict, constant.getRec, r)
0 users tested
Score so far: 0
100 users tested
Score so far: 77.97155778327935
200 users tested
Score so far: 163.24761858534592
300 users tested
Score so far: 247.0700325949549
400 users tested
```

```
Score so far: 329.3602045603418
500 users tested
Score so far: 413.3631291487538
600 users tested
Score so far: 498.4268891693222
700 users tested
Score so far: 581.1334125793506
800 users tested
Score so far: 663.7179944721634
900 users tested
Score so far: 746.0474962705288
AUC: 0.828894041097185
```

These are all evaluation metrics. They work the same for all recommendation algorithms. You just need to provide the getRec function.

The second non-personalized algorithm is the random recommender. You can use it like this

```
>>> randomRec = recommender.nonpersonalized.randomRec(trainingDict,
    1234567890)
>>> randomRec.getRec(0, 10)
[1548, 1068, 746, 1357, 1501, 1359, 428, 141, 233, 726]
>>> util.test.hitrate(evaluationDict, randomRec.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 2.0
200 users tested
Hits so far: 3.0
300 users tested
Hits so far: 3.0
400 users tested
Hits so far: 4.0
500 users tested
Hits so far: 4.0
600 users tested
Hits so far: 4.0
700 users tested
Hits so far: 4.0
800 users tested
Hits so far: 5.0
900 users tested
Hits so far: 5.0
Number of hits: 5.0
Number of possible hits: 943.0
Hitrate: 0.005302226935312832
```

The second paratemer of the constructor is the seed for the random function.

From now on we will only show the hitrate because the other metric are computed similarly. But feel free to also compute the other metrics as well.

7.3 k-nearest neighbor

Now we want to try the first more sophisticated algorithm, namely the item based k-Nearest Neighbor. The k-Nearest Neighbor algorithms need the database as a matrix so we have to first split the matrix provided by the reader object.

```
>>> trainingMatrix, matrixEvaluationDict = util.split.splitMatrix(r.
    getMatrix, 123456789)
0 Users split.
100 Users split.
200 Users split.
300 Users split.
400 Users split.
500 Users split.
600 Users split.
700 Users split.
800 Users split.
900 Users split.
```

Depending on the recommendation algorithm we need a matrix or a dict. So here we pass a matrix like r.getMatrix() returns and get a trainingMatrix where one entry per user is set to 0 and a dict with these missing entries. To understand the matrix represention of the dataset refer to 4.2.1

So we initialize an itemKnn object. The initialization builds the model yet so it will need some time until the constructor has finished.

```
import recommender.knn
itemKnn = recommender.knn.itemKnn(trainingMatrix, 10)
0 Similarities calculated
10000 Similarities calculated
20000 Similarities calculated
30000 Similarities calculated
...
1400000 Similarities calculated
1410000 Similarities calculated
```

Please note that you pass the the right dict for evaluation as the first parameter in this case it is matrixEvaluationDict. The second parameter is the number of neighbors to include into the neighborhood. During the model building phase the algorithm will compute the similarity of each item to each other item. But because this similarity is associative it is only necessary to compute the half. The 100k movieLens dataset We use for these examples has 1682 items so we get about $\frac{N(N-1)}{2} = \frac{1682^2}{2} = 1413721$ similarities to compute.

Now you can test this algorithm with the test metrics for example the hitrate

```
>>> util.test.hitrate(matrixEvaluationDict, itemKnn.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 21.0
```

```
200 users tested
Hits so far: 54.0
300 users tested
Hits so far: 78.0
400 users tested
Hits so far: 97.0
500 users tested
Hits so far: 126.0
600 users tested
Hits so far: 152.0
700 users tested
Hits so far: 181.0
800 users tested
Hits so far: 211.0
900 users tested
Hits so far: 239.0
Number of hits: 248.0
Number of possible hits: 943.0
Hitrate: 0.26299045599151644
```

As you can see this algorithm achieves much better performance than the non-personalized algorithms.

For user based k-Nearest Neighbors the call to build the model looks like this

```
>>> userKnn = recommender.knn.userKnn(trainingMatrix, 50)
0 Similarities calculated
10000 Similarities calculated
20000 Similarities calculated
30000 Similarities calculated
...
430000 Similarities calculated
440000 Similarities calculated
```

The dataset has 943 users so this time we have to compute approximately $\frac{N(N-1)}{2} = \frac{943*942}{2} = 444153$ similarities. To get the hitrate for this algorithm just like before we call the hitrate function with the getRec function.

```
>>> util.test.hitrate(matrixEvaluationDict, userKnn.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 21.0
200 users tested
Hits so far: 52.0
300 users tested
Hits so far: 82.0
400 users tested
Hits so far: 102.0
500 users tested
Hits so far: 130.0
600 users tested
Hits so far: 153.0
700 users tested
```

```
Hits so far: 173.0
800 users tested
Hits so far: 206.0
900 users tested
Hits so far: 237.0
Number of hits: 247.0
Number of possible hits: 943.0
Hitrate: 0.26193001060445387
0.26193001060445387
```

7.4 BPRMF

For the matrix factorization algorithms like BPRMF, RankMFX and Ranking SVD you need to specify some meta parameters.

```
>>> import recommender.BPRMF
>>> W, H = recommender.BPRMF.learnModel(r.getMaxUid(), r.getMaxIid(),
                        0.01, 0.01, 0.01,   # regularization parameter
                        0.1,                # learning rate
                        trainingDict,       # training dict
                        150,                # number of features
                        3,                  # number of epochs
                        r.numberOfTransactions)
Epoch: 1/3 | iteration 1000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.697530
Epoch: 1/3 | iteration 2000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.694108
Epoch: 1/3 | iteration 3000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.695815
...
Epoch: 3/3 | iteration 99000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.124050
Epoch: 3/3 | iteration 100000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.125070
```

This will output quite a much so you can observe how the loss is behaving. With a short search we found these values for the regularization parameters and the learning rate to be quite good, although they are not probably the best. But above all the results will get better with more epochs but then it will naturally take longer to compute so three is alright for a short experiment.

The learnModel returns a model in the form of two matrices. Two get a getRec method you have to instantiate a Mfrec class from the mf module in recommender.mf which offers such a method.

```
>>> BPRMF = recommender.mf.MFrec(W, H, trainingDict)
>>> util.test.hitrate(evaluationDict, BPRMF.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 26.0
200 users tested
```

```
Hits so far: 51.0
300 users tested
Hits so far: 78.0
400 users tested
Hits so far: 95.0
500 users tested
Hits so far: 122.0
600 users tested
Hits so far: 148.0
700 users tested
Hits so far: 171.0
800 users tested
Hits so far: 203.0
900 users tested
Hits so far: 224.0
Number of hits: 235.0
Number of possible hits: 943.0
Hitrate: 0.2492046659597031
```

As noted above the results get better with more epochs. As you can see in 5

7.5 RANKMFX

```
>>> import recommender.RankMFX
>>> W, H = recommender.RankMFX.learnModel(r.getMaxUid(), r.getMaxIid()
    ,
                    0.01, 0.01, 0.01, # regularization parameter
                    0.1,              # learning rate
                    trainingDict,     # training dict
                    250,              # number of features
                    5,                # number of epochs
                    r.numberOfTransactions)
Epoch: 1/5 | iteration 1000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.984394
Epoch: 1/5 | iteration 2000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.978983
Epoch: 1/5 | iteration 3000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.966176
...
Epoch: 5/5 | iteration 99000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.141662
Epoch: 5/5 | iteration 100000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.130552
```

Again there is much output to monitor the algorithm.

As above we have to generate a getRec method. Before we can evaluate the algorithm.

```
>>> RankMFX = recommender.mf.MFrec(W, H, trainingDict)
>>> util.test.hitrate(evaluationDict, RankMFX.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
```

```
Hits so far: 20.0
200 users tested
Hits so far: 37.0
300 users tested
Hits so far: 54.0
400 users tested
Hits so far: 73.0
500 users tested
Hits so far: 103.0
600 users tested
Hits so far: 123.0
700 users tested
Hits so far: 136.0
800 users tested
Hits so far: 154.0
900 users tested
Hits so far: 172.0
Number of hits: 177.0
Number of possible hits: 943.0
Hitrate: 0.18769883351007424
```

What applied to BPRMF in terms of performance also applies to RankMFX: With better tuned meta parameters and more time the results will likely be better.

7.6 Ranking SVD (Sparse SVD)

Next the Ranking SVD or Sparse SVD algorithm. To build the model we have a method similar to RankMFX and BPRMF, only without regularization parameters

```
>>> import recommender.svd
>>> W, H = recommender.svd.learnModel(r.getMaxUid(), r.getMaxIid(),
                    0.0002,         # learning rate
                    trainingDict,   # training dict
                    770,           # number of features
                    40,              # number of epochs
                    1000)           # number of iterations
Epoch: 1/40 | iteration 100/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.451531 | time needed:
     0.420000
Epoch: 1/40 | iteration 200/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.687917 | time needed:
     0.400000
Epoch: 1/40 | iteration 300/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.433482 | time needed:
     0.410000
...
Epoch: 40/40 | iteration 900/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -2.934971 | time needed:
     0.410000
```

```
Epoch: 40/40 | iteration 1000/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -2.734519 | time needed:
     0.400000
```

The evaluation works the same as for the other matrix factorization algorithms

```
>>> svd = recommender.mf.MFrec(W, H, trainingDict)
>>> util.test.hitrate(evaluationDict, svd.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 7.0
200 users tested
Hits so far: 19.0
300 users tested
Hits so far: 29.0
400 users tested
Hits so far: 37.0
500 users tested
Hits so far: 54.0
600 users tested
Hits so far: 71.0
700 users tested
Hits so far: 77.0
800 users tested
Hits so far: 95.0
900 users tested
Hits so far: 108.0
Number of hits: 113.0
Number of possible hits: 943.0
Hitrate: 0.11983032873807
```

Lastly the slopeone recommender

```
import recommender.slopeone
slopeone = recommender.slopeone.slopeone(trainingDict)
100000 differences computed
200000 differences computed
300000 differences computed
...
19800000 differences computed
19900000 differences computed
```

And its evaluation

```
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 11.0
200 users tested
Hits so far: 21.0
300 users tested
Hits so far: 28.0
400 users tested
```

```
Hits so far: 38.0
500 users tested
Hits so far: 49.0
600 users tested
Hits so far: 61.0
700 users tested
Hits so far: 70.0
800 users tested
Hits so far: 76.0
900 users tested
Hits so far: 78.0
Number of hits: 81.0
Number of possible hits: 943.0
Hitrate: 0.08589607635206786
```

8 CONCLUSIONS

In this document we provided an overview over the research area by explaining the terminology of recommender systems respectively collaborative filtering and their task. We also introduced the leave-one-out protocol used to evaluate the performance of recommender systems.

This is followed by explanations of the recommender algorithms implemented in recsyslab together with the core part of their implemention. By comparing the pseudocode and the Python code implementing it we can see that the implementation is quite simple.

After we provided the necessary foreknowledge we showed how the described algorithms perform with the also described metrics and we showed a high level view on recsyslab to help developers extend it.

All this should provide enough knowledge for the reader to understand roughly how recsyslab works in theory. So we finally got to the extensive user manual explaining how to use every recommender algorithm and test metric so the reader would be able to use the library.

The user manual shows that it is really easy to use recsyslab. Together with the comparisons of the Python code with the pseudocode in chapter 4 we see that the goals of this thesis are achieved.

While recsyslab provides several state of the art recommender algorithms, several widely used test metrics, a simple infrastructure to use train and test recommender algorithms. Additionally recsyslab is easy extendable with new recommender algorithms or test metrics and it produces acceptable test results After reading this document every beginner should be able to use and understand recsyslab.

We hope this will help students start working in this field and researchers to compare their new algorithms with already existing algorithms without wasting much time implementing something else then their own algorithms. Which was the motivation to write this thesis.

8.1 OUTLOOK

For future work we could implement more recommender algorithms, implement more test metrics or visualize test results graphically. It would also be handy to update the model with new interactions so we do not have to train the model from ground up again, or the possibility to continue training models after they got evaluated.

Also we want to find out why some algorithms does not perform so well. We suspect that there are errors in the implementation of slopeone and Ranking SVD. To find and correct this errors is certainly a future task. Also RankMFX is not performing as well as expected but there it is perhaps only the problem of finding the right parameters and training the model long enough, which will also be interesting to find out.

But apart of that we hope to get feedback from users to improve recsyslab. It will be interesting to see in which parts students have problems to understand what is going on. When such parts are discovered we want to make them easier to understand. Also which new features, test metrics and recommender algorithms get implemented will depend on the user feedback. For the recommender algorithms we will observe the research in this area and implement new popular recommender algorithms to keep recsyslab up to date.

9 APPENDIX - DOCSTRINGS

In this chapter we provide the inline documentation which is also included in recsyslab.

```
Help on package bin:

NAME
    bin - Package with the Python scripts of the recsyslab toolbox.

FILE
    /home/jk/recsyslab/bin/__init__.py

DESCRIPTION
    It contains the Packages:
        recommender -- Package for the recommender algorithms
        util        -- Package with all the other stuff needed to try
            out the
                        recommender algorithms

PACKAGE CONTENTS
    experiments
    main
    manual
    recommender (package)
    testing
    usecases
    util (package)
```

```
Help on package bin.recommender in bin:

NAME
    bin.recommender - Package containing the recommending algorithms.

FILE
    /home/jk/recsyslab/bin/recommender/__init__.py

DESCRIPTION
    These are:
        nonpersonalized --  constant and random
        knn             --  k-Nearest-Neigbor
        mf              --  Matrix Factorization with an arbitrary
            loss
        BPRMF           --  Bayesian Personalized Ranking with MF
                            (just calls mf with logLoss)
        RankMFX         --  MF with hingeLoss
        slopeone        --  slopeone recommender
        svd             --  Ranking SVD (Sparse SVD)
```

```
PACKAGE CONTENTS
    BPRMF
    RankMFX
    knn
    mf
    nonpersonalized
    slopeone
    svd
```

```
Help on module bin.recommender.knn in bin.recommender:

NAME
    bin.recommender.knn - Two classes for k-Nearest-Neighbor(knn)
        recommendation.

FILE
    /home/jk/recsyslab/bin/recommender/knn.py

DESCRIPTION
    itemKnn --  Item based knn
    userKnn --  User based knn

    Based on:
    George Karypis. 2001. Evaluation of Item-Based Top-N
        Recommendation
    Algorithms. In Proceedings of the tenth international conference
        on
    Information and knowledge management (CIKM 01), Henrique Paques,
    Ling Liu, and David Grossman (Eds.). ACM, New York, NY, USA,
        247-254.
    DOI=10.1145/502585.502627 http://doi.acm.org/10.1145/502585.502627

CLASSES
    __builtin__.object
        itemKnn
        userKnn

    class itemKnn(__builtin__.object)
     |  Class for item based knn.
     |
     |  Methods defined here:
     |
     |  __init__(self, userItemMatrix, n)
     |      Builds a model for Item based knn.
     |
     |          userItemMatrix  --  A matrix where an entry at i, j is
          the rating
     |                              the ith user gave the jth item.
     |          n               --  number of neighbors which are
         getting computed.
     |
     |      Uses the cosine for similarity.
     |
     |  getRec(self, u, n)
     |      Returns the n best recommendations for user u.
     |
     |      Set n = -1 to get all items recommended
     |
     |  ------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

    class userKnn(__builtin__.object)
     |  Methods defined here:
     |
     |  __init__(self, userItemMatrix, n)
     |      Builds a model for user based knn.
     |
     |          userItemMatrix  --  A matrix where an entry at i, j is
          the rating
     |                              the ith user gave the jth item.
     |          n               --  number of neighbors which get
         computed.
     |
     |      Uses the cosine for similarity.
     |
     |  getRec(self, u, n)
     |      Returns the n best recommendations for user u.
     |
     |      Set n = -1 to get all items recommended
     |
     |  ------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    computeCosSim(sim, matrix)
        Computes the cos of every two lines in the matrix and writes
            them into sim.

    cos(a, b)
        Gets two vectors(onedimensional np.matrix) and computes their
            cosine.
```

```
Help on module bin.recommender.mf in bin.recommender:

NAME
    bin.recommender.mf - Module for Matrix Factorization (mf)
        techniques.

FILE
    /home/jk/recsyslab/bin/recommender/mf.py

DESCRIPTION
    learnModel  --  learn a mf model
    MFrec       --  compute recommendations based on a mf model

CLASSES
    __builtin__.object
        MFrec

    class MFrec(__builtin__.object)
     |  Class to compute recommendations with a mf model.
     |
     |  Methods defined here:
     |
     |  __init__(self, W, H, trainingR)
     |      Initialize the class.
     |
     |      W           --  Matrix of the User features
     |      H           --  Matrix of the Item features
     |      trainingR   --  The dict the model was trained with
     |
     |  getRec(self, u, n)
     |      Returns the n best recommendations for user u based on the
          mf model.
     |
     |      Set n = -1 to get all items recommended
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    learnModel(n_users, m_items, regU, regI, regJ, learningRate, R,
        features, epochs, numberOfIterations, lossF, dlossF)
        Learns a mf model with a passed loss function.

            n_users             --  The highest internal assigned User
                 ID
            m_items             --  The highest internal assigned Item
                 ID
            regU                --  Regularization for the user vector
            regI                --  Regularization for the positive
                item
            regJ                --  Regularization for the negative
                item
            learningRate        --  The learning rate
            R                   --  A dict of the form UserID -> (
                ItemId, Rating)
            features            --  Number of features of the items
                and users
            epochs              --  Number of epochs the model should
                be learned
            numberOfIterations  --  Number of iterations in each epoch
            lossF               --  Loss function
            dlossF              --  Derivation of lossF

        Returns:
            W   --  User Features
            H   --  Item Features

        Based on:
        Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and
        Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking
            from
        implicit feedback. In Proceedings of the Twenty-Fifth
            Conference on
        Uncertainty in Artificial Intelligence (UAI 09).
        AUAI Press, Arlington, Virginia, United States, 452-461.
```

```
Help on module bin.recommender.BPRMF in bin.recommender:

NAME
    bin.recommender.BPRMF

FILE
    /home/jk/recsyslab/bin/recommender/BPRMF.py

FUNCTIONS
    dLogLoss(a, y)
        Derivative of the logLoss.

    learnModel(n_users, m_items, regU, regI, regJ, learningRate, R,
        features, epochs, numberOfIterations)
        Learns a model with the BPRMF algorithm, actually just calls
            mf.

            n_users             --  The highest internal assigned User
                 ID
            m_items             --  The highest internal assigned Item
                 ID
            regU                --  Regularization for the user vector
            regI                --  Regularization for the positive
                item
            regJ                --  Regularization for the negative
                item
            learningRate        --  The learning rate
            R                   --  A dict of the form UserID -> (
                ItemId, Rating)
            features            --  Number of features of the items
                and users
            epochs              --  Number of epochs the model should
                be learned
            numberOfIterations  --  Number of iterations in each epoch

        Returns:
            W   --  User Features
            H   --  Item Features

        Based on:
        Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and
        Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking
            from
        implicit feedback. In Proceedings of the Twenty-Fifth
            Conference on
        Uncertainty in Artificial Intelligence (UAI 09).
        AUAI Press, Arlington, Virginia, United States, 452-461.

    logLoss(a, y)
        logLoss(a, y) = log(1 + exp(-a*y))
```

```
Help on module bin.recommender.RankMFX in bin.recommender:

NAME
    bin.recommender.RankMFX

FILE
    /home/jk/recsyslab/bin/recommender/RankMFX.py

FUNCTIONS
    dHingeLoss(a, y)
        Derivative of the hingeLoss.

    hingeLoss(a, y)
        hingeLoss(a, y) = max(0, 1 - a*y)

    learnModel(n_users, m_items, regU, regI, regJ, learningRate, R,
        features, epochs, numberOfIterations)
        Learns a model with the RankMFX algorithm, actually just calls
             mf.

            n_users             --  The highest internal assigned User
                 ID
            m_items             --  The highest internal assigned Item
                 ID
            regU                --  Regularization for the user vector
            regI                --  Regularization for the positive
                item
            regJ                --  Regularization for the negative
                item
            learningRate        --  The learning rate
            R                   --  A dict of the form UserID -> (
                ItemId, Rating)
            features            --  Number of features of the items
                and users
            epochs              --  Number of epochs the model should
                be learned
            numberOfIterations  --  Number of iterations in each epoch

        Returns:
            W   --  User Features
            H   --  Item Features
```

```
Help on module bin.recommender.slopeone in bin.recommender:

NAME
    bin.recommender.slopeone

FILE
    /home/jk/recsyslab/bin/recommender/slopeone.py

CLASSES
    __builtin__.object
        slopeone

    class slopeone(__builtin__.object)
     |  A class to compute recommendations with a Slope One Predictor.
     |
     |  Based on:
     |      "Slope One Predictors for Online Rating-Based
         Collaborative Filtering"
     |      by Daniel Lemire and Anna Maclachlan.
     |
     |  Methods defined here:
     |
     |  __init__(self, R)
     |      Generate the model.
     |
     |      R   --  A dict of the form UserID -> (ItemId, Rating)
     |
     |  getRec(self, u, n)
     |      Returns the n best recommendations for user u.
     |
     |      Set n = -1 to get all items recommended
     |
     |  ------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
```

```
Help on module bin.recommender.svd in bin.recommender:

NAME
    bin.recommender.svd - Module to compute the model of Ranking SVD (
        Sparse SVD)

FILE
    /home/jk/recsyslab/bin/recommender/svd.py

DESCRIPTION
    Based on:
    Jahrer, Michael, and Andreas Toescher.
    "Collaborative filtering ensemble for ranking."
    Proc. of KDD Cup Workshop at 17th ACM SIGKDD Int. Conf.
    on Knowledge Discovery and Data Mining, KDD. Vol. 11. 2011.

FUNCTIONS
    learnModel(n_users, m_items, learningRate, R, features, epochs,
        numberOfIterations)
        Returns the model of a learned svd as two matrices.

        n_users             --  The highest internal assigned User ID
        m_items             --  The highest internal assigned Item ID
        learningRate        --  The learning rate
        R                   --  A dict of the form UserID -> (ItemId,
            Rating)
        features            --  Number of features of the items and
            users
        epochs              --  Number of epochs the model should be
            learned
        numberOfIterations  --  Number of iterations in each epoch
```

```
Help on package bin.util in bin:

NAME
    bin.util - Package with utility code to run the recommender
        algorithms.

FILE
    /home/jk/recsyslab/bin/util/__init__.py

DESCRIPTION
    It contains the following modules:
        reader  --  A reader to read in databases
        split   --  Split databases
        test    --  Test metrics
        helper  --  Several helping methods and a two way dict
```

```
PACKAGE CONTENTS
    helper
    reader
    split
    test
```

```
Help on module bin.util.reader in bin.util:

NAME
    bin.util.reader - Contains a class to manage a database.

FILE
    /home/jk/recsyslab/bin/util/reader.py

CLASSES
    __builtin__.object
        stringSepReader

    class stringSepReader(__builtin__.object)
     |  A class to manage a database.
     |
     |  Methods defined here:
     |
     |  __init__(self, filename, separator)
     |      Reads in a database file.
     |
     |      The lines of the database file have to look like this:
     |          UserID<separator>ItemID<separator>Rating
     |      If there is just an UserID and an ItemID the rating is set
      to 1.
     |      Everything coming after the rating will be ignored,
     |      but when omit the rating and still have something
         following the ItemID
     |      this will be understood as the rating.
     |
     |  getInternalIid(self, originalIid)
     |      Maps the given original ItemID to the corresponding
         internal ItemID.
     |
     |      The passed original UserID has to be a string.
     |
     |  getInternalUid(self, originalUid)
     |      Maps the given original UserID to the corresponding
         internal UserID.
     |
     |      The passed original UserID has to be a string.
     |
     |  getMatrix(self)
     |      Get the database as a matrix.
     |
     |      The lines of the matrix are corresponding to the users
     |      and the columns to the items.
     |      So the n,m entry is the rating the nth user gave the mth
         item.
     |
     |  getMaxIid(self)
     |      Returns the highest internal assigned ItemID.
     |
     |  getMaxUid(self)
     |      Returns the highest internal assigned UserID.
     |
     |  getOriginalIid(self, internalIid)
     |      Maps the given internal ItemID to the corresponding
         original ItemID.
     |
     |  getOriginalUid(self, internalUid)
     |      Maps the given internal UserID to the corresponding
         original UserID.
     |
     |  getR(self)
     |      Return the database as a dict.
     |
     |      The dict has internal UserIDs as keys and
     |      (ItemID, Rating) Tuples as values
     |
     |  ------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
```

```
Help on module bin.util.split in bin.util:

NAME
    bin.util.split - Methods to split a database with the leave-one-
        out method.

FILE
    /home/jk/recsyslab/bin/util/split.py

DESCRIPTION
    split        --  splits a reader into two new dicts
    splitMatrix  --  splits a matrix into a matrix and a dict

FUNCTIONS
    split(R, seed)
        Splits a database into two dicts.

        Splits after the leave-one-out method which means for every
            user in the
        database it takes one item out of the database and into a new
            one.
        The first returned dict is the database with one item missing
            for each
        user. The second returned dict has the missing items.

        R is a dict of the database.
        seed is a seed for the randomness.

    splitMatrix(M, seed)
        Splits a matrix into two new dicts.

        Returns an User x Item Matrix with one entry of each user set
            to 0
        and a User -> Item Dict with the missing entrys.

        M is an User x Item Matrix.
        seed is a seed for the randomness.
```

```
Help on module bin.util.test in bin.util:

NAME
    bin.util.test - Module with several metrics to test the
        recommender algorithms.

FILE
    /home/jk/recsyslab/bin/util/test.py

DESCRIPTION
    hitrate     --  Returns #hits, #items
    precision   --  Returns #hits / n
    f1          --  Returns the result of a F1 test
    mrhr        --  Returns the Mean Reciprocal Hitrate
    auc         --  Returns the Area under the curve
    countHits   --  Returns hits, items in a tuple

FUNCTIONS
    auc(testR, recommender, r)
        Returns and prints the AUC of the recommender function.

        testR is a dict with an internal UserID as a dict and a list
            of items as
        values. Normally testR is the second dict split.split returns.
        The list can have a lenght greater than 1.

        recommender is a function which takes an internal UserID and n
             and returns
        n items recommender for the UserID.

        r is a reader object with the database read in.

        Based on:
        Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and
        Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking
            from
        implicit feedback. In Proceedings of the Twenty-Fifth
            Conference on
        Uncertainty in Artificial Intelligence (UAI 09).
        AUAI Press, Arlington, Virginia, United States, 452-461.

    countHits(testR, recommender, n)
        Returns the number of hits and the number of items it was
            tested with.

        Is a helper function for the other metrics
        testR is a dict with an internal UserID as a dict and a list
            of items as
        values. Normally testR is the second dict split.split returns.
        The list can have a lenght greater than 1.

        recommender is a function which takes an internal UserID and n
             and returns
        n items recommender for the UserID.

        n is the number of items the recommender can recommend.

        hits = number of items from testR the recommender guessed
            right.
        items = the number of items in testR.

        hits, items will be printed.
        hits, items will get returned.

    f1(testR, recommender, n)
        Prints and returns F1 of the recommender.

        F1 = (2 * hitrate * precision) / (hitrate + precision)
        testR is a dict with an internal UserID as a dict and a list
            of items as
        values. Normally testR is the second dict split.split returns.
        The list can have a lenght greater than 1.

        recommender is a function which takes an internal UserID and n
             and returns
        n items recommender for the UserID.

        n is the number of items the recommender can recommend.

        hits = number of items from testR the recommender guessed
            right.

        See also: "Application of Dimensionality Reduction in
            Recommender System
        -- A Case Study" by Badrul M. sarcar et al.

    hitrate(testR, recommender, n)
        Returns the number of hits and the number of items it was
            tested with.
        testR is a dict with an internal UserID as a dict and a list
            of items as
        values. Normally testR is the second dict split.split returns.
        The list can have a lenght greater than 1.

        recommender is a function which takes an internal UserID and n
             and returns
        n items recommender for the UserID.

        n is the number of items the recommender can recommend.

        hits = number of items from testR the recommender guessed
            right.
        items = the number of items in testR.

        hits, items and hits / items will be printed.
        hits, items will get returned.

        hits / items gives the hitrate or recall.

        See also:
        Sarwar, Badrul, et al. Application of dimensionality reduction
             in
        recommender system-a case study. No. TR-00-043.
        MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE, 2000.

    mrhr(testR, recommender, n)
        Returns the Mean Reciprocal Hitrate of the recommender.

        testR is a dict with an internal UserID as a dict and a list
            of items as
        values. Normally testR is the second dict split.split returns.
        The list can have a length greater than 1.

        recommender is a function which takes an internal UserID and n
             and returns
        n items recommender for the UserID.

        n is the number of items the recommender can recommend.

        hits = number of items from testR the recommender guessed
            right.

    precision(testR, recommender, n)
        Returns the number of hits / n .

        testR is a dict with an internal UserID as a dict and a list
            of items as
        values. Normally testR is the second dict split.split returns.
        The list can have a lenght greater than 1.
```

```
        recommender is a function which takes an internal UserID and n
             and returns
        n items recommender for the UserID.

        n is the number of items the recommender can recommend.

        hits = number of items from testR the recommender guessed
            right.

        See also:
        Sarwar, Badrul, et al. Application of dimensionality reduction
             in
        recommender system-a case study. No. TR-00-043.
        MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE, 2000.
```

```
Help on module bin.util.helper in bin.util:

NAME
    bin.util.helper - Several helper functions and a helper class.

FILE
    /home/jk/recsyslab/bin/util/helper.py

DESCRIPTION
    idOrigDict          --  two way dict
    normRowNorm         --  normalize a matrix by the norm
    normRowSum          --  normalize a matrix by the sum
    cache               --  cache results of a function
    writeOrigToFile     --  save a DB with original IDs
    writeInternalToFile --  save a DB with internal IDs
    sortList            --  sort a list with scores and IDs
    listToSet           --  convert a list into a set
    sortResults         --  sort test results
    printMatrix         --  print a matrix
    randDataset         --  generate a random DB
    dictToMatrix        --  convert a dict to a matrix
    getScoreMF          --  compute the score of an item
    getExternalRec      --  convert getRec for original IDs

CLASSES
    __builtin__.object
        idOrigDict

    class idOrigDict(__builtin__.object)
     |  Class two map external/original IDs to internal IDs and vice
         versa.
     |
     |  Methods defined here:
     |
     |  __init__(self)
     |      Initialization.
     |
     |  add(self, orig)
     |      Add a new original ID.
     |
     |      Returns the internal ID the passed original ID got mapped
         to.
     |      If the passed ID is already mapped nothing happens except
         the
     |      mapped internal ID is returned.
     |
     |  getId(self, orig)
     |      Returns the internal ID which is mapped to the passed
         orinigal ID.
     |
     |  getOrig(self, id)
     |      Returns the original ID which is mapped to the passed
         internal ID.
     |
     |
         ----------------------------------------------------------------------

     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    cache(fn)
        Decorator to cache the result of the function fn.

    dictToMatrix(d)
        Converts a dict like reader.getR() returns to a matrix.

    getExternalRec(getRec, r)
        Converts getRec so it takes and returns only the original IDs.

        r   --  reader object

    getScoreMF(origUid, origIid, W, H, r)
        Returns the score of one item for one user with an MF model.
        origUid --  original User ID
        origIid --  original Item ID
        W, H    --  The MF model like returned by recommender.mf for
            example
        r       --  a reader object with the database

    listToSet(sortedscorelist, n)
        Converts a list into a set with size n.

        Gets a list like sortList returns and returns a set with the
            first n
        items, or less, when there are not enough items
```

```
normRowNorm(m)
    Normalize the matrix in place so each row of the matrix has
        norm 1.

normRowSum(m)
    Normalize the matrix in place so each row of the matrix has
        sum 1.

printMatrix(m)
    Prints a whole matrix.

    Prints the whole matrix m without replacing stuff with dots
    like it normally does when the matrix is to large.

randDataset(user, items, p, seed, filename)
    Generates a random dataset and writes it into a file.

    user    --  number of users
    items   --  number of items
    p       --  percentage of ones in the dataset
    seed    --  seed for the randomness

sortList(scorelist)
    Gets a list of tuples (itemid, score), sorts by score
        decreasing.

sortResults(name)
    Sort test results written to a file.

writeInternalToFile(reader, toSave, filename)
    Writes a database into a file with the internal IDs.

        reader  --  reader object
        toSave  --  Part of the database to write
        filename--  Name of the file

    Writes toSave into a file with the name filename.

writeOrigToFile(reader, toSave, filename)
    Writes a database into a file with the original IDs.

        reader  --  reader object
        toSave  --  Part of the database to write
        filename--  Name of the file

    Writes toSave into a file with the name filename but first the
    internal IDs in toSave get mapped to the original IDs.
```

BIBLIOGRAPHY

[1] Cofirank, June 2013. URL https://sites.google.com/a/cofirank.org/index/Home.

[2] Collaborative filtering, June 2013. URL http://en.wikipedia.org/wiki/Collaborative_filtering.

[3] Content-based filtering, June 2013. URL http://en.wikipedia.org/wiki/Recommender_system#Content-based_filtering.

[4] Duine framework, June 2013. URL http://www.duineframework.org/.

[5] The leave-one-out protocol, June 2013. URL http://en.wikipedia.org/wiki/Cross-validation_%28statistics%29#Leave-one-out_cross-validation.

[6] Apache mahout, June 2013. URL http://mahout.apache.org/.

[7] Matrix factorization, June 2013. URL http://en.wikipedia.org/wiki/Matrix_factorization.

[8] Movielens data sets, June 2013. URL http://grouplens.org/node/73.

[9] Netflix prize, June 2013. URL http://www.netflixprize.com/.

[10] Numpy, June 2013. URL http://www.numpy.org/.

[11] Python programming language, June 2013. URL http://www.python.org/.

[12] recsyslab, June 2013. URL https://github.com/Foolius/recsyslab.

[13] Snap twitter dataset, June 2013. URL http://snap.stanford.edu/data/twitter7.html.

[14] User based knn, June 2013. URL https://github.com/zenogantner/MyMediaLite/blob/master/src/MyMediaLite/RatingPrediction/UserKNN.cs.

[15] Youtube video platform, June 2013. URL https://www.youtube.com/.

[16] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[17] Mukund Deshpande and George Karypis. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems (TOIS)*, 22(1):143–177, 2004.

[18] Ernesto Diaz-Aviles, Lucas Drumond, Zeno Gantner, Lars Schmidt-Thieme, and Wolfgang Nejdl. What is happening right now... that interests me?: online topic discovery and recommendation in twitter. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1592–1596. ACM, 2012.

[19] Michael D. Ekstrand, Michael Ludwig, Joseph A. Konstan, and John T. Riedl. Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit. In *Proceedings of the fifth ACM conference on Recommender systems*, RecSys '11, pages 133–140, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0683-6. doi: 10.1145/2043932.2043958. URL http://doi.acm.org/10.1145/2043932.2043958.

[20] Zeno Gantner, Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. MyMediaLite: A free recommender system library. In *Proceedings of the 5th ACM Conference on Recommender Systems (RecSys 2011)*, 2011.

[21] Michael Jahrer and Andreas Töscher. Collaborative filtering ensemble for ranking. In *Proc. of KDD Cup Workshop at 17th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, KDD*, volume 11, 2011.

[22] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *Proceedings of the tenth international conference on Information and knowledge management*, CIKM '01, pages 247–254, New York, NY, USA, 2001. ACM. ISBN 1-58113-436-3. doi: 10.1145/502585.502627. URL http://doi.acm.org/10.1145/502585.502627.

[23] J. Lee, M. Sun, and G. Lebanon. A Comparative Study of Collaborative Filtering Algorithms. *ArXiv e-prints*, May 2012.

[24] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering. *CoRR*, abs/cs/0702144, 2007.

[25] Xia Ning and George Karypis. Slim: Sparse linear methods for top-n recommender systems. In Diane J. Cook, Jian Pei, Wei Wang, Osmar R. Zaïane, and Xindong Wu, editors, *ICDM*, pages 497–506. IEEE, 2011. ISBN 978-0-7695-4408-3.

[26] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 452–461, Arlington, Virginia, United States, 2009. AUAI Press. ISBN 978-0-9749039-5-8. URL http://dl.acm.org/citation.cfm?id=1795114.1795167.

[27] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John T. Riedl. Application of dimensionality reduction in recommender system – a case study. In *IN ACM WEBKDD WORKSHOP*, 2000.