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.

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 [9] in the programming language Python [8]. 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 [9] in der Programmiersprache Python [8] 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 [12]. 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 [6] 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's usual in rating prediction scenarios because it's normally not useful to predict explicit absolute ratings when you want to propose items a user could find interesting. In this case it's 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

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. Followed by a chapter with results and explanations about how the results have been obtained. After that we'll 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 techniques used to evaluate recommender algorithms.

2.1 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 [3]. You can use the Leave-one-out protocol with many different metrics which are also explained here.

2.1.1 *Leave-one-out Protocol*

The Leave-one-out protocol means, that we take one interaction of each user out of the database for training and use it for validation. The item the recommender has to predict is also called hidden item. Now we can test for each user if the algorithm is capable to predict this missing interaction. Most of the test metrics described below are counting how many of the hidden items the algorithm recommends while being restricted to only recommend a certain number of items. But all of them take recommendations for each user. This wouldn't be possible if we would for example just take out 1% of the interactions it's likely that there are some users without a hidden item. Then the metrics wouldn't work anymore.

2.1.2 *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.

2.1.2.1 *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 [18, 23]. 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 doesn't take the number of recommended items into account.

2.1.2.2 *Precision*

The precision [23] 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.

2.1.2.3 *F1*

The F1 metric [23] tries to balance hitrate and precision by taking both into account.

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

2.1.2.4 *Mean Reciprocal Hitrate*

The mean reciprocal hitrate or more general mean reciprocal rank [21] 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 $\text{pos}(\text{topN}_u, H_u)$ is the position of the hidden item in the recommendation.

2.1.2.5 *Area under the ROC (AUC)*

AUC [22] 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.

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

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

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

2.2 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.

2.2.1 *MovieLens*

MovieLens [5] 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.

2.2.2 *Million Song Dataset*

The million song dataset [13] 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's easier for others to retrace the results.

2.2.3 *476 Million Twitter Tweets Dataset*

The Stanford Network Analysis Project provided a twitter dataset with about 467 million tweets from 17.000 users [10]. 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 [16] 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 [19] 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 [4] 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 [2] 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 [15] 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

4 RECOMMENDATION ALGORITHMS

In this chapter I will provide an overview on how the algorithms I've implemented work. 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 doesn't 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.

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.

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 [18] two similarity functions are compared. Cosines performs best so in recsyslab only the cosine is implemented. For two vectors $\vec{v}, \vec{u}$ The cosine is defined as

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

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.

To compute recommendations for user U the algorithm then 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 [18].

4.2.2 *User Based*

The user based k-Nearest-Neighbor [11] 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.

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.

4.3 MATRIX FACTORIZATION

All matrix factorization techniques build two matrices in the model building phase. These matrices 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. The process of presenting a large matrix M as two smaller matrices W and H so that $M = W \times H$ is also called singular value decomposition.

Each of the implemented algorithms train the model with stochastic gradient descent. In each iteration the model is trained with a randomly chosen user, a randomly chosen item the user interacted with, called the positive item and a randomly chosen item the user didn't interacted with yet, called the negative item. The features of the user and the negative and the positive item are then trained according to the derivative of a loss function.

4.3.1 *BPRMF*

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

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

And the derivative of the log loss is

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

For further informations please refer to [22]

4.3.2 *RankMFX*

RankMFX uses the hingeLoss. It is defined as

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

And its derivative

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

See also [14].

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 [17].

4.3.4 *Slope One*

The Slope One recommendation algorithm computes the differences of interaction intensity between items and uses these differences to predict indirectly the interaction intensity between users and items without any interaction yet [20].

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. 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 chech it out for further implementation details.

5.2 RESULTS

recommender	hitrate	precision	f1	mrhr	auc
constant	0.0731707317073	6.9	0.14480587618	0.0207687387433	0.826383065429
randomRec	0.00530222693531	0.5	0.0104931794334	0.00110463061152	0.479019845932
itemKnn	0.00530222693531	0.5	0.0104931794334	0.00207670554966	0.56107542212
userKnn	0.0084835630965	0.8	0.0167890870934	0.00382054907506	0.53129413142
BPRMF	0.24920466596	23.5	0.493179433368	0.0996574593075	0.929450479792
RankMFX	0.18769883351	17.7	0.371458551941	0.0852135198371	0.867928869256
Ranking SVD	0.00742311770944	0.7	0.0146904512067	0.00162138733189	0.517177605189
slopeone	0.0	0.0	0	0.0	0.669577426536

5.3 COMPARISON

6 DESIGN AND IMPLEMENTATION

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

6.1 general structure

One important part is the reader class in the util package. 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's 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 hasn't 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.

6.2 interfaces

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 [8] and NumPy [7]. 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 4. For informations about the test metrics refer to 2.1.2.

The following python code is also in the file manual.py in the bin directory of the library so you don't have to copy the code out of this pdf document. But if you're working through these examples please note that you perhaps miss the import of a module when you skip an example because We won't 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're 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

To load the dataset it has to be a textfile where each line is of the following format:

```
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 and one will be assumed. Everything coming after NumberOfInteractions<separator> will be ignored. Please note that when you're omitting NumberOfInteractions but have something else after the ItemId, this will be recognized as NumberOfInteractions.

We recommend to use the MovieLens database2.2.1 with 100,000 ratings. It is easy to get, doesn't need any modifications to work with our library and has a reasonable size. Also we will use this dataset in the following examples. When you have a suiting database, start up a python interpreter of version 2.7.x.

[1] We will use $ to indicate a bash prompt.

```
$ 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.
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 2.1.1. 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.
```

[2] We will use >>> to indicate the python prompt.

```
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 don't 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
>>> hits, items = 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

```
>>> 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

```
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

```
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 don't 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 don't 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's 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's 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, 10)
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: 45.0
300 users tested
Hits so far: 72.0
400 users tested
Hits so far: 91.0
500 users tested
Hits so far: 120.0
600 users tested
Hits so far: 142.0
700 users tested
Hits so far: 164.0
800 users tested
Hits so far: 196.0
900 users tested
Hits so far: 223.0
Number of hits: 230.0
Number of possible hits: 943.0
Hitrate: 0.24390243902439024
```

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 aren't 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
                    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 = 1.000484
Epoch: 1/3 | iteration 2000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.987704
Epoch: 1/3 | iteration 3000/100000 | learning rate=0.100000 |
    average_loss for the last 1000 iterations = 0.986371
```

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, BPRMF.getRec, 10)
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 14.0
200 users tested
Hits so far: 32.0
300 users tested
Hits so far: 54.0
400 users tested
Hits so far: 71.0
500 users tested
Hits so far: 89.0
600 users tested
Hits so far: 105.0
700 users tested
Hits so far: 120.0
800 users tested
Hits so far: 138.0
900 users tested
Hits so far: 152.0
Number of hits: 162.0
Number of possible hits: 943.0
Hitrate: 0.17179215270413573
```

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
                    1000,            # number of features
                    1,               # number of epochs
                    1000)            # number of iterations
Epoch: 1/1 | iteration 100/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.713911 | time needed:
     0.560000
Epoch: 1/1 | iteration 200/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.507542 | time needed:
     0.580000
Epoch: 1/1 | iteration 300/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.626713 | time needed:
     0.610000
```

```
Epoch: 1/1 | iteration 400/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.717652 | time needed:
     0.570000
Epoch: 1/1 | iteration 500/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.535933 | time needed:
     0.580000
Epoch: 1/1 | iteration 600/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.572086 | time needed:
     0.580000
Epoch: 1/1 | iteration 700/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.682302 | time needed:
     0.580000
Epoch: 1/1 | iteration 800/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.731191 | time needed:
     0.580000
Epoch: 1/1 | iteration 900/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.507198 | time needed:
     0.580000
Epoch: 1/1 | iteration 1000/1000 | learning rate=0.000200 |
    average_loss for the last 100 iterations = -3.663757 | time needed:
     0.570000
```

The evaluation works the same as for the other 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: 1.0
200 users tested
Hits so far: 2.0
300 users tested
Hits so far: 2.0
400 users tested
Hits so far: 2.0
500 users tested
Hits so far: 3.0
600 users tested
Hits so far: 4.0
700 users tested
Hits so far: 6.0
800 users tested
Hits so far: 6.0
900 users tested
Hits so far: 6.0
Number of hits: 7.0
Number of possible hits: 943.0
Hitrate: 0.007423117709437964
```

Lastly the slopeone recommender

```
import recommender.slopeone
slopeone = recommender.slopeone.slopeone(trainingDict)
100000 differences computed
```

```
200000 differences computed
300000 differences computed
...
20000000 differences computed
20100000 differences computed
```

And its evaluation

```
0 users tested
Hits so far: 0.0
100 users tested
Hits so far: 0.0
200 users tested
Hits so far: 0.0
300 users tested
Hits so far: 0.0
400 users tested
Hits so far: 0.0
500 users tested
Hits so far: 0.0
600 users tested
Hits so far: 0.0
700 users tested
Hits so far: 0.0
800 users tested
Hits so far: 0.0
900 users tested
Hits so far: 0.0
Number of hits: 0.0
Number of possible hits: 943.0
Hitrate: 0.0
# I really hope we can get better
```

8 CONCLUSIONS

In this document I provided

- an overview over the research area
- explanations of the implemented recommender algorithms
- explanations of the implemented test metrics
- a user manual explaining how to use every recommonder algorithm and test metric
- explanations of the internal structure of recsyslab
- test results

While recsyslab provides

- several state of the art recommender algorithms
- several widely used test metrics
- a simple infrastructure to use these together
- easy extendable with new recommender algorithms or test metrics
- 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.

8.1 OUTLOOK

For future work we could

- implement more recommender algorithms
- implement more test metrics
- visualize test results graphically
- update the model with new interactions so we don't have to train the model from ground up again

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's 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

```
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

See also: "BPR: Bayesian Personalized Ranking from Implicit
Feedback" from Steffen Rendle et al.
```

BIBLIOGRAPHY

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

[2] Duine framework. URL http://www.duineframework.org/.

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

[4] Apache mahout. URL http://mahout.apache.org/.

[5] Movielens data sets. URL http://grouplens.org/node/73.

[6] Netflix prize. URL http://www.netflixprize.com/.

[7] Numpy. URL http://www.numpy.org/.

[8] Python programming language. URL http://www.python.org/.

[9] recsyslab. URL https://github.com/Foolius/recsyslab.

[10] Snap twitter dataset. URL http://snap.stanford.edu/data/twitter7.html.

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

[12] Youtube video platform. URL https://www.youtube.com/.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.