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 [?] in the programming language Python [?]. 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 [?] in der Programmiersprache Python [?] 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 [**?**]. 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 [**?**] 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

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 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 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 [**?**]. 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 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.

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 [**?** **?**]. 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.

2.1.2.2 *Precision*

The precision [**?**] 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 [**?**] 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 [**?**] 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.

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

Where N is the number of items in the dataset and $\mathrm{pos}(\mathrm{topN}_u, H_u)$ is the position of the hidden item in the recommendation.

2.1.2.5 *Area under the ROC (AUC)*

AUC [?] 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

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

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 [?] 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 [**?**] 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.

2.2.3 *476 Million Twitter Tweets Dataset*

The Stanford Network Analysis Project provided a twitter dataset with about 467 million tweets from 17.000 users [**?**]. 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 [?] 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 [?] 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 [?] 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 [?] 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 [?] 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 [?] 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 have implemented work. And show the core part of their source code in recsyslab to illustrate the simplicity of the code. 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 [**?**] 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:

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

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
```

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 [**?**]. 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
\end{lstlisting



\subsection{User Based}

The user based k-Nearest-Neighbor~\cite{userbasedknn} 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.



\section{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 did not
    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.



\subsection{BPRMF}

BPMRF uses the logloss to train the model. The logloss is defined
as
\[
\textrm{logLoss}(a,y)=\log(1+\exp(-ay))
\]



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~\cite{Rendle:2009:BBP
    :1795114.1795167}
```

```
\subsection{RankMFX}

RankMFX uses the hingeLoss. It is defined as

\[
\mathrm{\textrm{hingeLoss}(a,y)=\max(0,1-ay)}
\]


And its derivative

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


See also~\cite{diaz2012happening}.


\subsection{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~\cite{jahrer2011collaborative}.

\subsection{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~\cite{DBLP:journals/corr/abs-cs-0702144}.
```