

Specialist Programme on Artificial Intelligence for IT & ITES Industry

Recommender Systems Workshops

Dr Barry Shepherd
barryshepherd@nus.edu.sg

Singapore e-Government Leadership Centre
National University of Singapore

© 2020 NUS. The contents contained in this document may not be reproduced in any form or by any means, without the written permission of ISS, NUS other than for the purpose for which it has been supplied.

Inspire *Lead* *Transform*

Workshop 1

- Explore some simple code that implements User-based and Item-based Collaborative Filtering and also implements basic testing
- I will walk you through the code using a simple dataset
- Then your task:
 - Apply the code to 2 larger datasets:
 - ❖ Movielens (movie ratings)
 - ❖ Jester (joke ratings)
 - ❖ BookCrossings (book ratings)
 - Compare the performance of User-based versus Item-based CF
 - Compare the performance of different similarity measures
 - Compare the performance with user-normalised ratings versus un-normalised

Workshop1: Datasets

- **Movielens** = Ratings from 1 to 5 (has 1M, 10M, 20M datasets)
- **Jester** = Ratings from -10 to +10 (~6M ratings of 150 jokes)
- **Book-Crossing** = Ratings from 1 to 10 (~1.1M ratings of 270K books)

Dataset	Users	Items	Ratings	Density	Rating Scale
➡ Movielens 1M	6040	3883	1,000,209	4.26%	[1-5]
Movielens 10M	69,878	10,681	10,000,054	1.33%	[0.5-5]
Movielens 20M	138,493	27,278	20,000,263	0.52%	[0.5-5]
➡ Jester	124,113	150	5,865,235	31.50%	[-10, 10]
➡ Book-Crossing	92,107	271,379	1,031,175	0.0041%	[1, 10], and implicit
Last.fm	1892	17632	92,834	0.28%	Play Counts
Wikipedia	5,583,724	4,936,761	417,996,366	0.0015%	Interactions
OpenStreetMap (Azerbaijan)	231	108,330	205,774	0.82%	Interactions
Git (Django)	790	1757	13,165	0.95%	Interactions




9 Must-Have Datasets for Investigating Recommender Systems

<https://www.kdnuggets.com/2016/02/nine-datasets-investigating-recommender-systems.html>




MovieLens Website

To get started, tell MovieLens about your preferences by distributing 3 points among your favorite groups of movies below.

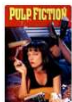


computer animation, good versus evil, mythology

+  Toy Story  The Lord of the Rings: The Fellowship of the Ring  Harry Potter and the Philosopher's Stone



dramatic, good acting, intense

+  Forrest Gump  Million Dollar Baby  The Social Network




blood, dark humor, social commentary

+  Pulp Fiction  Kill Bill: Vol. 1  American History X



action, fun movie, special effects

+  True Lies  The Mask

chick flick, feel-good, touching

+  Titanic  Dead Poets Society  Slumdog Millionaire

classic, masterpiece, quotable

+  The Godfather  Psycho

<https://movielens.org/home>

Community Tags

view:

- | | | |
|--|---|---|
| <input type="button" value="x260"/> Morgan Freeman + | <input type="button" value="x214"/> prison + | <input type="button" value="x204"/> prison escape + |
| <input type="button" value="x154"/> friendship + | <input type="button" value="x134"/> Stephen King + | <input type="button" value="x127"/> classic + |
| <input type="button" value="x89"/> justice + | <input type="button" value="x81"/> great acting + | <input type="button" value="x83"/> reflective + |
| <input type="button" value="x54"/> heartwarming + | <input type="button" value="x60"/> Tim Robbins + | <input type="button" value="x54"/> imdb top 250 + |
| <input type="button" value="x37"/> redemption + | <input type="button" value="x42"/> crime + | <input type="button" value="x33"/> sentimental + |
| <input type="button" value="x20"/> good story + | <input type="button" value="x19"/> great performances + | <input type="button" value="x19"/> inspiring + |
| <input type="button" value="x14"/> prison drama + | <input type="button" value="x16"/> mystery + | <input type="button" value="x13"/> clever + |
| <input type="button" value="x13"/> violence + | <input type="button" value="x12"/> corruption + | <input type="button" value="x12"/> intelligent + |
| <input type="button" value="x11"/> must see + | <input type="button" value="x10"/> excellent script + | <input type="button" value="x8"/> existentialism + |

top picks

The Shawshank Redemption

1994 R 142 min



★★★★★

Schindler's List

1993 R 195 min



★★★★★

The Godfather

1972 R 175 min



★★★★★

The Dark Knight

2008 PG-13 152 min



★★★★★

The Matrix

1999 136 min




★★★★★

recent releases

Hellboy


2019



★★★★★

After


2019



★★★★★

Little


2019



★★★★★

The Head Hunter


2019 72 min



★★★★★

The Best of Enemies

2019



★★★★★

MovieLens Data Set (100K ratings)

- Each user has rated at least 20 movies
- The data is randomly ordered. Users & items are numbered consecutively from 1.
- Ratings are made on a 5-star scale (integer only)
- Timestamp is represented in seconds since 1/1/1970 UTC

UserID	movie	rating	datetime
1	61	4	878542420
1	189	3	888732928
1	33	4	878542699
1	160	4	875072547
1	20	4	887431883
1	202	5	875072442
1	171	5	889751711
1	265	4	878542441

Ratings file

movie id	movie name	Action	Adventure	Animation	Children's
1	Toy Story (1995)	0	0	1	1
2	GoldenEye (1995)	1	1	0	0
3	Four Rooms (1995)	0	0	0	0
4	Get Shorty (1995)	1	0	0	0
5	Copcat (1995)	0	0	0	0
6	Shanghai Triad (Yac	0	0	0	0
7	Twelve Monkeys (1	0	0	0	0
8	Babe (1995)	0	0	0	1
9	Dead Man Walking	0	0	0	0

*Movie names
& Genres*

userid	age	gender	occupation	zip code
1	24	M	technician	85711
2	53	F	other	94043
3	23	M	writer	32067
4	24	M	technician	43537
5	33	F	other	15213
6	42	M	executive	98101

*User
demographics*

<https://grouplens.org/datasets/>

Jester Dataset

- Data collected from a joke recommendation website
 - Ratings are real values from -10.00 to +10.00
- Dataset1 (tabular format)
 - Ratings of 100 jokes
 - 24,983 users who have rated 36 or more jokes
 - Stored as a matrix (table)
 - "99" corresponds to "not rated"
- Dataset2 (transaction format)
 - Ratings of 100 jokes
 - 23,500 users who have rated 36 or more jokes

Sherlock Holmes and Dr. Watson go on a camping trip, set up their tent, and fall asleep. Some hours later, Holmes wakes his faithful friend. "Watson, look up at the sky and tell me what you see."


Watson replies, "I see millions of stars."

"What does that tell you?"

Watson ponders for a minute. "Astronomically speaking, it tells me that there are millions of galaxies and potentially billions of planets. Astrologically, it tells me that Saturn is in Leo. Timewise, it appears to be approximately a quarter past three. Theologically, it's evident the Lord is all-powerful and we are small and insignificant. Meteorologically, it seems we will have a beautiful day tomorrow. What does it tell you?"

Holmes is silent for a moment, then speaks. "Watson, you idiot, someone has stolen our tent."

Less Funny More Funny



Next

<http://eigentaste.berkeley.edu/dataset/>

BookCrossings Dataset

- Book Ratings from 1 to 10 (~1.1M ratings of 270K books)
 - Ratings from 1 to 10 are explicit (actual ratings)
 - Ratings = 0 are implicit (imply the user read the book) – ignore for now
- There are 3 files with this dataset but we use only the ratings for this workshop
 1. BX-Book-Ratings ~ the book ratings: *User.ID, ISBN, Book.Rating (transaction format)*
 2. BX-Users ~ demographic info: *UserID, Location, Age* (but many fields are blank)
 3. BX-Books ~ content info: *Title, Author, Publication year, Publisher*

```
In [381]: trans = pd.read_csv("BX-Book-Ratings.csv", sep=';', error_bad_lines=False,
encoding="latin-1")

In [382]: trans
Out[382]:
```

	User-ID	ISBN	Book-Rating
0	276725	034545104X	0
1	276726	0155061224	5
2	276727	0446520802	0
3	276729	052165615X	3
4	276729	0521795028	6
...
1149775	276704	1563526298	9
1149776	276706	0679447156	0
1149777	276709	0515107662	10
1149778	276721	0590442449	10
1149779	276723	05162443314	8

```
[1149780 rows x 3 columns]
```

Also see <https://www.bookcrossing.com/>

<http://www2.informatik.uni-freiburg.de/~ciegler/BX/>

Handling Very Sparse Data

- Storing the ratings in a full (uncompressed) matrix is impractical when the data is very sparse
- In the Bookcrossings dataset there are 270K books and 92K users
 - Matrix size = 270K * 92K ~ 24Billion cells, yet only ~ 1M ratings are given (0.004%)
 - Trying to create the full matrix fails... not enough memory

```
rtmatrix, uids, _iids = makeratingsmatrix(trans)
```



```
IndexError: index 1494529060 is out of bounds for axis 0 with size 1494492565
```

- A workaround is to data sample – this is shown in workshop1 demo code
 - We sample only the most popular books and the most active users
- A better solution is to use a **Sparse Matrix** representation for the ratings matrix
 - Store only the non-empty values, along with the cell indexes
 - Sparse matrices are used in the Surprise & Implicit Libraries
 - We will explore these in workshop2

Workshop1: Getting Started

File	Contains
demolib.py	<ul style="list-style-type: none"> Simple implementation of User-Based and Item-Based CF Employs a non-optimised data representation (simple array)
CF1A-workshop.py	<ul style="list-style-type: none"> Demo code to apply the above functions to the “Toby” dataset
simplemovies-transactions.csv	<ul style="list-style-type: none"> The “Toby” dataset
u_data.csv	<ul style="list-style-type: none"> Movielens 100K dataset
u_item.csv	<ul style="list-style-type: none"> Mapping of movieID to actual movie name (and release date , genre)
jester_ratings.txt	<ul style="list-style-type: none"> Jester dataset2
BX-Book-Ratings.csv	<ul style="list-style-type: none"> Book crossings dataset

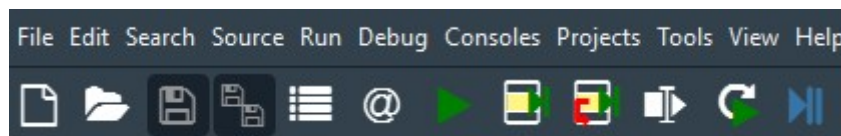
(1) Download the above files
 (2) Run Spyder and open:
demolib.py and *CF1Aworkshop.py*



(3) select the *demolib.py* tab and run the file



(4) select the *CF1A workshop* tab and then single step thro the code by running each line separately.



runs the file

runs current line
 (where the cursor is)

Remember to first change the working directory to the directory where you placed the data files using:

path = 'D:/datasets' (for example)

Some Benchmark Results

- See <https://www.librec.net/release/v1.3/example.html>

Rating Prediction: MovieLens 1M, 100K, Epinions, FilmTrust, Ciao, Flixster;

Item Ranking: MovieLens 100K, Epinions, Flixster, FilmTrust, Ciao;

MovieLens (100K)						
Algorithm	MAE			RMSE		
	MMLite	PREA	LibRec	MMLite	PREA	LibRec
GlobalAvg	0.945	0.949	0.945	1.126	1.128	1.126
UserAvg	0.835	0.838	0.835	1.041	1.043	1.042
ItemAvg	0.817	0.823	0.817	1.024	1.030	1.025
PD	N/A	N/A	0.794	N/A	N/A	1.094
	sigma=2.5					
UserKNN	0.721	0.732	0.737	0.921	0.937	0.944
	neighbors=60, shrinkage=25, similarity=pcc; MMLite: reg_u=12, reg_i=1					
ItemKNN	0.703	0.716	0.723	0.899	0.914	0.924
	neighbors=40, shrinkage=2500, similarity=pcc; MMLite: reg_u=12, reg_i=1					
SlopeOne	0.739	0.740	0.739	0.939	0.940	0.940
RegSVD	0.741	0.730	0.730	0.949	0.932	0.936
	factors=10, reg=0.05, learn.rate=0.005, max.iter=100					
BiasedMF	0.724	N/A	0.722	0.918	N/A	0.918

Rating Prediction

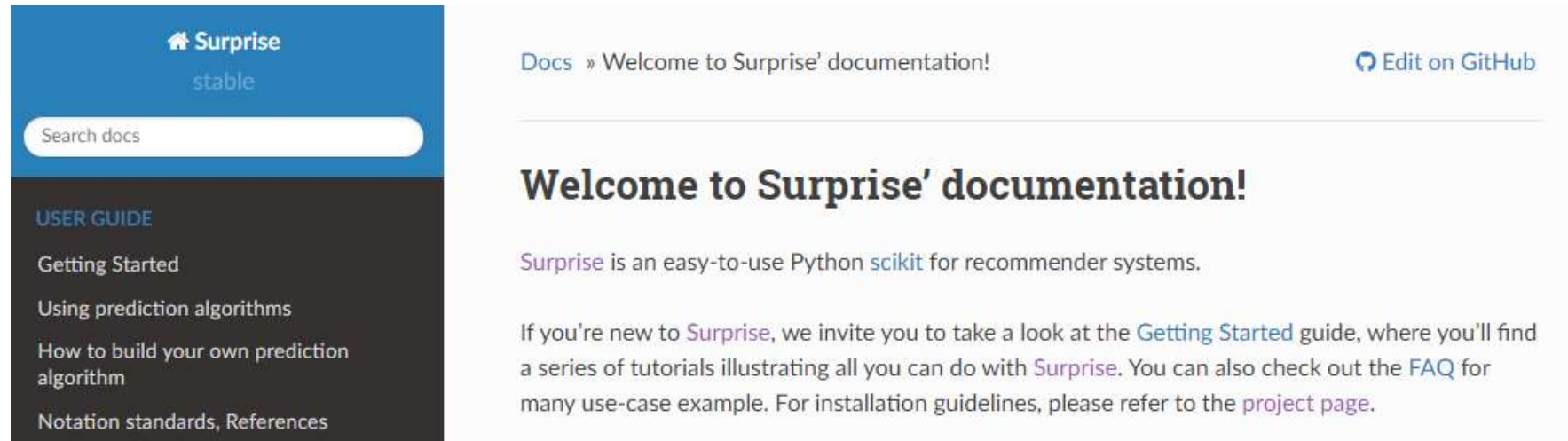
MovieLens (100K)						
Algo	Prec@5		Prec@10		Recall@5	
	MMLite	LibRec	MMLite	LibRec	MMLite	LibRec
MostPop	0.212	0.211	0.192	0.190	0.071	0.070
ItemKNN	0.314	0.318	0.279	0.260	0.096	0.103
	neighbors=80, similarity=cos, shrinkage=50, thresh=-1					
UserKNN	0.397	0.338	0.334	0.280	0.138	0.116
	neighbors=80, similarity=cos, shrinkage=50, thresh=-1					
BPR	0.358	0.378	0.309	0.321	0.247	0.129
	factors=10, reg=0.01, learn.rate=0.05, max.iter=30					
WRMF	0.416	0.424	0.353	0.358	0.142	0.149
	alpha=1.0, factors=20, reg=0.015, max.iter=10					

Item Ranking

Workshop2

- Explore the Surprise Library
- Use the demo code (surprise-demo.py) to apply
 - User-based and Item-based collaborative filtering
 - Matrix factorisation (SVD algorithm)
- Use
 - Movielens – as before
 - Bookcrossings – use ALL explicit ratings (no need to sub-sample)
- Compare the best MAE for each dataset with the results from workshop1A

Surprise Library – User Guide Extracts

A screenshot of the Surprise library documentation page. The left sidebar is dark blue with a 'Surprise stable' header, a search bar, and a 'USER GUIDE' section containing links to 'Getting Started', 'Using prediction algorithms', 'How to build your own prediction algorithm', and 'Notation standards, References'. The main content area is white and shows the 'Welcome to Surprise' documentation page, which includes a 'Docs » Welcome to Surprise' breadcrumb, an 'Edit on GitHub' link, and introductory text about the library.

<https://surprise.readthedocs.io/en/stable/>

Surprise Algorithms

<code>random_pred.NormalPredictor</code>	Algorithm predicting a random rating based on the distribution of the training set, which is assumed to be normal.
<code>baseline_only.BaselineOnly</code>	Algorithm predicting the baseline estimate for given user and item.
<code>knns.KNNBasic</code>	A basic collaborative filtering algorithm.
<code>knns.KNNWithMeans</code>	A basic collaborative filtering algorithm, taking into account the mean ratings of each user.
<code>knns.KNNWithZScore</code>	A basic collaborative filtering algorithm, taking into account
<code>knns.KNNBaseline</code>	A basic collaborative filtering algorithm taking into account a <i>baseline</i> rating.
<code>matrix_factorization.SVD</code>	The famous SVD algorithm, as popularized by Simon Funk during the Netflix Prize. When baselines are not used, this
<code>matrix_factorization.SVDpp</code>	The SVD++ algorithm, an extension of <code>svd</code> taking into account implicit ratings.
<code>matrix_factorization.NMF</code>	A collaborative filtering algorithm based on Non-negative Matrix Factorization.
<code>slope_one.SlopeOne</code>	A simple yet accurate collaborative filtering algorithm.
<code>co_clustering.CoClustering</code>	A collaborative filtering algorithm based on co-clustering.

<https://surprise.readthedocs.io/en/stable/>

Surprise: KNN Algorithms

k-NN inspired algorithms

These are algorithms that are directly derived from a basic nearest neighbors approach.

Note

For each of these algorithms, the actual number of neighbors that are aggregated to compute an estimation is necessarily less than or equal to k . First, there might just not exist enough neighbors and second, the sets $N_i^k(u)$ and $N_u^k(i)$ only include neighbors for which the similarity measure is **positive**. It would make no sense to aggregate ratings from users (or items) that are negatively correlated. For a given prediction, the actual number of neighbors can be retrieved in the `'actual_k'` field of the `details` dictionary of the `prediction`.

Surprise: User-based & Item-based CF

```
class surprise.prediction_algorithms.knns.KNNBasic(k=40, min_k=1, sim_options={}, verbose=True,
**kwargs)
```

A basic collaborative filtering algorithm.

The prediction \hat{r}_{ui} is set as:

User-based

$$\hat{r}_{ui} = \frac{\sum_{v \in N_i^k(u)} \text{sim}(u, v) \cdot r_{vi}}{\sum_{v \in N_i^k(u)} \text{sim}(u, v)}$$

or

Item-based

$$\hat{r}_{ui} = \frac{\sum_{j \in N_u^k(i)} \text{sim}(i, j) \cdot r_{uj}}{\sum_{j \in N_u^k(i)} \text{sim}(i, j)}$$

Parameters:

- **k** (int) – The (max) number of neighbors to take into account for aggregation (see [this note](#)). Default is `40`.
- **min_k** (int) – The minimum number of neighbors to take into account for aggregation. If there are not enough neighbors, the prediction is set to the global mean of all ratings. Default is `1`.
- **sim_options** (dict) – A dictionary of options for the similarity measure. See [Similarity measure configuration](#) for accepted options.
- **verbose** (bool) – Whether to print trace messages of bias estimation, similarity, etc. Default is True.

depending on the `user based` field of the `sim_options` parameter.

```
# compute similarities between items
sim_options = {'name': 'cosine',
               'user_based': False
               }
algo = KNNBasic(sim_options=sim_options)
```

Surprise: User-based & Item-based CF

Similarity measure configuration

Many algorithms use a similarity measure to estimate a rating. The way they can be configured is done in a similar fashion as for baseline ratings: you just need to pass a `sim_options` argument at the creation of an algorithm. This argument is a dictionary with the following (all optional) keys:

- `'name'`: The name of the similarity to use, as defined in the `similarities` module. Default is `'MSD'`.
- `'user_based'`: Whether similarities will be computed between users or between items. This has a **huge** impact on the performance of a prediction algorithm. Default is `True`.
- `'min_support'`: The minimum number of common items (when `'user_based'` is `'True'`) or minimum number of common users (when `'user_based'` is `'False'`) for the similarity not to be zero. Simply put, if $|I_{uv}| < \text{min_support}$ then $\text{sim}(u, v) = 0$. The same goes for items.
- `'shrinkage'`: Shrinkage parameter to apply (only relevant for `pearson_baseline` similarity). Default is 100.

Surprise: Similarity Measures

Available similarity measures:

<code>cosine</code>	Compute the cosine similarity between all pairs of users (or items).
<code>msd</code>	Compute the Mean Squared Difference similarity between all pairs of users (or items).
<code>pearson</code>	Compute the Pearson correlation coefficient between all pairs of users (or items).
<code>pearson_baseline</code>	Compute the (shrunk) Pearson correlation coefficient between all pairs of users (or items) using baselines for centering instead of means.

MSD Similarity (Euclidean)

`surprise.similarities.msd()`

Only **common** users (or items) are taken into account. The Mean Squared Difference is defined as:

$$\text{msd}(u, v) = \frac{1}{|I_{uv}|} \cdot \sum_{i \in I_{uv}} (r_{ui} - r_{vi})^2$$

or

$$\text{msd}(i, j) = \frac{1}{|U_{ij}|} \cdot \sum_{u \in U_{ij}} (r_{ui} - r_{uj})^2$$

depending on the `user_based` field of `sim_options` (see [Similarity measure configuration](#)).

The MSD-similarity is then defined as:

$$\begin{aligned} \text{msd_sim}(u, v) &= \frac{1}{\text{msd}(u, v) + 1} \\ \text{msd_sim}(i, j) &= \frac{1}{\text{msd}(i, j) + 1} \end{aligned}$$

The +1 term is just here to avoid dividing by zero.

Pearson Similarity

`surprise.similarities.pearson()`

Compute the Pearson correlation coefficient between all pairs of users (or items).

Only **common** users (or items) are taken into account. The Pearson correlation coefficient can be seen as a mean-centered cosine similarity, and is defined as:

$$\text{pearson_sim}(u, v) = \frac{\sum_{i \in I_{uv}} (r_{ui} - \mu_u) \cdot (r_{vi} - \mu_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - \mu_u)^2} \cdot \sqrt{\sum_{i \in I_{uv}} (r_{vi} - \mu_v)^2}}$$

or

$$\text{pearson_sim}(i, j) = \frac{\sum_{u \in U_{ij}} (r_{ui} - \mu_i) \cdot (r_{uj} - \mu_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \mu_i)^2} \cdot \sqrt{\sum_{u \in U_{ij}} (r_{uj} - \mu_j)^2}}$$

depending on the `user_based` field of `sim_options` (see [Similarity measure configuration](#)).

Note: if there are no common users or items, similarity will be 0 (and not -1).

Pearson-Baseline Similarity

`surprise.similarities.pearson_baseline()`

Compute the (shrunk) Pearson correlation coefficient between all pairs of users (or items) using baselines for centering instead of means.

The shrinkage parameter helps to avoid overfitting when only few ratings are available (see [Similarity measure configuration](#)).

The Pearson-baseline correlation coefficient is defined as:

$$\text{pearson_baseline_sim}(u, v) = \hat{\rho}_{uv} = \frac{\sum_{i \in I_{uv}} (r_{ui} - b_{ui}) \cdot (r_{vi} - b_{vi})}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - b_{ui})^2} \cdot \sqrt{\sum_{i \in I_{uv}} (r_{vi} - b_{vi})^2}}$$

or

$$\text{pearson_baseline_sim}(i, j) = \hat{\rho}_{ij} = \frac{\sum_{u \in U_{ij}} (r_{ui} - b_{ui}) \cdot (r_{uj} - b_{uj})}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - b_{ui})^2} \cdot \sqrt{\sum_{u \in U_{ij}} (r_{uj} - b_{uj})^2}}$$

The shrunk Pearson-baseline correlation coefficient is then defined as:

$$\begin{aligned} \text{pearson_baseline_shrunk_sim}(u, v) &= \frac{|I_{uv}| - 1}{|I_{uv}| - 1 + \text{shrinkage}} \cdot \hat{\rho}_{uv} \\ \text{pearson_baseline_shrunk_sim}(i, j) &= \frac{|U_{ij}| - 1}{|U_{ij}| - 1 + \text{shrinkage}} \cdot \hat{\rho}_{ij} \end{aligned}$$

Surprise: User-based & Item-based CF

```
class surprise.prediction_algorithms.knns.KNNWithMeans(k=40, min_k=1, sim_options={},
verbose=True, **kwargs)
```

A basic collaborative filtering algorithm, taking into account the mean ratings of each user.

The prediction \hat{r}_{ui} is set as:

$$\hat{r}_{ui} = \mu_u + \frac{\sum_{v \in N_i^k(u)} \text{sim}(u, v) \cdot (r_{vi} - \mu_v)}{\sum_{v \in N_i^k(u)} \text{sim}(u, v)}$$

or

$$\hat{r}_{ui} = \mu_i + \frac{\sum_{j \in N_u^k(i)} \text{sim}(i, j) \cdot (r_{uj} - \mu_j)}{\sum_{j \in N_u^k(i)} \text{sim}(i, j)}$$

depending on the `user_based` field of the `sim_options` parameter.

Parameters:

- **k** (int) - The (max) number of neighbors to take into account for aggregation (see [this note](#)). Default is `40`.
- **min_k** (int) - The minimum number of neighbors to take into account for aggregation. If there are not enough neighbors, the neighbor aggregation is set to zero (so the prediction ends up being equivalent to the mean μ_u or μ_i). Default is `1`.
- **sim_options** (dict) - A dictionary of options for the similarity measure. See [Similarity measure configuration](#) for accepted options.
- **verbose** (bool) - Whether to print trace messages of bias estimation, similarity, etc. Default is True.

`KNNBasic(sim_options={'name': 'pearson'})` \neq `KNNWithMeans(sim_options={'name': 'pearson'})`

MF Algorithms

```
class surprise.prediction_algorithms.matrix_factorization.SVD
```

The famous SVD algorithm, as popularized by [Simon Funk](#) during the Netflix Prize. When baselines are not used, this is equivalent to Probabilistic Matrix Factorization [SM08] (see [note](#) below).

The prediction \hat{r}_{ui} is set as:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

If user u is unknown, then the bias b_u and the factors p_u are assumed to be zero. The same applies for item i with b_i and q_i .

To estimate all the unknown, we minimize the following regularized squared error:

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda (b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2)$$

The minimization is performed by a very straightforward stochastic gradient descent:

$$\begin{aligned} b_u &\leftarrow b_u + \gamma(e_{ui} - \lambda b_u) \\ b_i &\leftarrow b_i + \gamma(e_{ui} - \lambda b_i) \\ p_u &\leftarrow p_u + \gamma(e_{ui} \cdot q_i - \lambda p_u) \\ q_i &\leftarrow q_i + \gamma(e_{ui} \cdot p_u - \lambda q_i) \end{aligned}$$

where $e_{ui} = r_{ui} - \hat{r}_{ui}$. These steps are performed over all the ratings of the trainset and repeated `n_epochs` times. Baselines are initialized to `0`. User and item factors are randomly initialized according to a normal distribution,

You also have control over the learning rate γ and the regularization term λ . Both can be different for each kind of parameter (see below). By default, learning rates are set to `0.005` and regularization terms are set to `0.02`.

MF Parameters

- **n_factors** – The number of factors. Default is `100`.
- **n_epochs** – The number of iteration of the SGD procedure. Default is `20`.
- **biased** (*bool*) – Whether to use baselines (or biases). See [note](#) above. Default is `True`.
- **init_mean** – The mean of the normal distribution for factor vectors initialization. Default is `0`.
- **init_std_dev** – The standard deviation of the normal distribution for factor vectors initialization. Default is `0.1`.
- **lr_all** – The learning rate for all parameters. Default is `0.005`.
- **reg_all** – The regularization term for all parameters. Default is `0.02`.

- **lr_bu** – The learning rate for b_u .
- **lr_bi** – The learning rate for b_i .
- **lr_pu** – The learning rate for p_u .
- **lr_qi** – The learning rate for q_i .

- **reg_bu** – The regularization term for b_u .
- **reg_bi** – The regularization term for b_i .
- **reg_pu** – The regularization term for p_u .
- **reg_qi** – The regularization term for q_i .

You can choose to use an unbiased version of this algorithm, simply predicting:

$$\hat{r}_{ui} = q_i^T p_u$$

This is equivalent to Probabilistic Matrix Factorization ([SM08], section 2) and can be achieved by setting the **biased** parameter to `False`.

- **random_state** (int, RandomState instance from numpy, or `None`) – Determines the RNG that will be used for initialization. If int, `random_state` will be used as a seed for a new RNG. This is useful to get the same initialization over multiple calls to `fit()`. If RandomState instance, this same instance is used as RNG. If `None`, the current RNG from numpy is used. Default is `None`.

Workshop3 – Implicit Ratings

- Explore the Implicit Library
 - Use the Bookcrossings data set (implicit ratings only)
 - Use the Deskdrop dataset (new)
- Use the demo code to apply
 - Matrix factorisation for Implicit data (ALS algorithm)
 - Bayesian Personalised Ranking (if time)



<https://www.kaggle.com/gspmoreira/articles-sharing-reading-from-cit-deskdrop>

DeskDrop Dataset

- Deskdrop is an internal communications platform that allows company employees to share relevant articles with their peers, and collaborate around them.
- Dataset size:
 - 12 months logs (Mar. 2016 - Feb. 2017) from DeskDrop with ~73k logged users interactions on more than 3k public articles shared in the platform.
- Dataset content:
 - Item attributes: Articles' original URL, title, and content plain text are available in two languages (English and Portuguese).
 - Contextual information: Context of the users visits, like date/time, client (mobile native app / browser) and geolocation.
 - Logged users: All users are required to login in the platform, providing a long-term tracking of users preferences (not depending on cookies in devices).
 - Rich implicit feedback: Different interaction types were logged, making it possible to infer the user's level of interest in the articles (eg. comments > likes > views).
 - Multi-platform: Users interactions were tracked in different platforms (web browsers and mobile native apps)

Implicit Library – User Guide Extracts

Implicit

Fast Python Collaborative Filtering for Implicit Datasets

This project provides fast Python implementations of several different popular recommendation algorithms for implicit feedback datasets:

- Alternating Least Squares as described in the papers [Collaborative Filtering for Implicit Feedback Datasets](#) and in [Applications of the Conjugate Gradient Method for Implicit Feedback Collaborative Filtering](#).
- [Bayesian Personalized Ranking](#)
- Item-Item Nearest Neighbour models, using Cosine, TFIDF or BM25 as a distance metric

All models have multi-threaded training routines, using Cython and OpenMP to fit the models in parallel among all available CPU cores. In addition, the ALS and BPR models both have custom CUDA kernels - enabling fitting on compatible GPU's. This library also supports using approximate nearest neighbours libraries such as [Annoy](#), [NMSLIB](#) and [Faiss](#) for speeding up making recommendations.

<https://implicit.readthedocs.io/en/latest/quickstart.html>

Implicit Library – User Guide Extracts

Basic Usage

```
import implicit

# initialize a model
model = implicit.als.AlternatingLeastSquares(factors=50)

# train the model on a sparse matrix of item/user/confidence weights
model.fit(item_user_data)

# recommend items for a user
user_items = item_user_data.T.tocsr()
recommendations = model.recommend(userid, user_items)

# find related items
related = model.similar_items(itemid)
```

Not ratings!

AlternatingLeastSquares

```
class implicit.als.AlternatingLeastSquares(factors=100, regularization=0.01, dtype=<type
'numpy.float32'>, use_native=True, use_cg=True, use_gpu=False, iterations=15, calculate_training_loss=False,
num_threads=0, random_state=None)
```

A Recommendation Model based off the algorithms described in the paper 'Collaborative Filtering for Implicit Feedback Datasets' with performance optimizations described in 'Applications of the Conjugate Gradient Method for Implicit Feedback Collaborative Filtering.'

- Parameters:**
- **factors** (*int, optional*) – The number of latent factors to compute
 - **regularization** (*float, optional*) – The regularization factor to use
 - **dtype** (*data-type, optional*) – Specifies whether to generate 64 bit or 32 bit floating point factors
 - **use_native** (*bool, optional*) – Use native extensions to speed up model fitting
 - **use_cg** (*bool, optional*) – Use a faster Conjugate Gradient solver to calculate factors
 - **use_gpu** (*bool, optional*) – Fit on the GPU if available, default is to run on GPU only if available
 - **iterations** (*int, optional*) – The number of ALS iterations to use when fitting data
 - **calculate_training_loss** (*bool, optional*) – Whether to log out the training loss at each iteration
 - **num_threads** (*int, optional*) – The number of threads to use for fitting the model. This only applies for the native extensions. Specifying 0 means to default to the number of cores on the machine.
 - **random_state** (*int, RandomState or None, optional*) – The random state for seeding the initial item and user factors. Default is None.

BayesianPersonalizedRanking

```
class implicit.bpr.BayesianPersonalizedRanking
```

Bayesian Personalized Ranking

A recommender model that learns a matrix factorization embedding based off minimizing the pairwise ranking loss described in the paper [BPR: Bayesian Personalized Ranking from Implicit Feedback](#).

- Parameters:**
- **factors** (*int, optional*) – The number of latent factors to compute
 - **learning_rate** (*float, optional*) – The learning rate to apply for SGD updates during training
 - **regularization** (*float, optional*) – The regularization factor to use
 - **dtype** (*data-type, optional*) – Specifies whether to generate 64 bit or 32 bit floating point factors
 - **use_gpu** (*bool, optional*) – Fit on the GPU if available
 - **iterations** (*int, optional*) – The number of training epochs to use when fitting the data
 - **verify_negative_samples** (*bool, optional*) – When sampling negative items, check if the randomly picked negative item has actually been liked by the user. This check increases the time needed to train but usually leads to better predictions.
 - **num_threads** (*int, optional*) – The number of threads to use for fitting the model. This only applies for the native extensions. Specifying 0 means to default to the number of cores on the machine.
 - **random_state** (*int, RandomState or None, optional*) – The random state for seeding the initial item and user factors. Default is None.

```
explain(userid, user_items, itemid, user_weights=None, N=10)
```

Provides explanations for why the item is liked by the user.

Parameters:

- **userid** (*int*) – The userid to explain recommendations for
- **user_items** (*csr_matrix*) – Sparse matrix containing the liked items for the user
- **itemid** (*int*) – The itemid to explain recommendations for
- **user_weights** (*ndarray, optional*) – Precomputed Cholesky decomposition of the weighted user liked items. Useful for speeding up repeated calls to this function, this value is returned
- **N** (*int, optional*) – The number of liked items to show the contribution for

Returns:

- **total_score** (*float*) – The total predicted score for this user/item pair
- **top_contributions** (*list*) – A list of the top N (itemid, score) contributions for this user/item pair
- **user_weights** (*ndarray*) – A factorized representation of the user. Passing this in to future 'explain' calls will lead to noticeable speedups