# LensKit Documentation

***Release 0.5.0***

**Michael D. Ekstrand**

**Jan 14, 2019**

# Contents:

LensKit is a set of Python tools for experimenting with and studying recommender systems. It provides support for training, running, and evaluating recommender algorithms in a flexible fashion suitable for research and education.

LensKit for Python (also known as LKPY) is the successor to the Java-based LensKit project.

# CHAPTER 1

# Installation

To install the current release with Anaconda (recommended):

```
conda install -c lenskit lenskit
```

Or you can use `pip`:

```
pip install lenskit
```

To use the latest development version, install directly from GitHub:

```
pip install git+https://github.com/lenskit/lkpy
```

Then see Getting Started.

# CHAPTER 2

# Resources

- Mailing list, etc.
- Source and issues on GitHub

## 2.1 Getting Started

This notebook gets you started with a brief nDCG evaluation with LensKit for Python.

### 2.1.1 Setup

We first import the LensKit components we need:

[1]:
```
from lenskit import batch, topn, util
from lenskit import crossfold as xf
from lenskit.algorithms import als, item_knn as knn
from lenskit.metrics import topn as tnmetrics
```

And Pandas is very useful:

[2]:
```
import pandas as pd
```

[3]:
```
%matplotlib inline
```

### 2.1.2 Loading Data

We're going to use the ML-100K data set:

[4]:
```
ratings = pd.read_csv('ml-100k/u.data', sep='\t',
                      names=['user', 'item', 'rating', 'timestamp'])
ratings.head()
```

[4]:
```
   user  item  rating  timestamp
0   196   242       3  881250949
1   186   302       3  891717742
2    22   377       1  878887116
3   244    51       2  880606923
4   166   346       1  886397596
```

### 2.1.3 Defining Algorithms

Let's set up two algorithms:

[5]:
```
algo_ii = knn.ItemItem(20)
algo_als = als.BiasedMF(50)
```

### 2.1.4 Running the Evaluation

In LensKit, our evaluation proceeds in 2 steps:

1. Generate recommendations
2. Measure them

If memory is a concern, we can measure while generating, but we will not do that for now.

We will first define a function to generate recommendations from one algorithm over a single partition of the data set. It will take an algorithm, a train set, and a test set, and return the recommendations.

**Note:** before fitting the algorithm, we clone it. Some algorithms misbehave when fit multiple times.

The code function looks like this:

[6]:
```
def eval(aname, algo, train, test):
    fittable = util.clone(algo)
    algo.fit(train)
    users = test.user.unique()
    # the recommend function can merge rating values
    recs = batch.recommend(algo, users, 100,
            topn.UnratedCandidates(train), test)
    # add the algorithm
    recs['Algorithm'] = aname
    return recs
```

Now, we will loop over the data and the algorithms, and generate recommendations:

[7]:
```
all_recs = []
test_data = []
for train, test in xf.partition_users(ratings[['user', 'item', 'rating']], 5, xf.
↪SampleFrac(0.2)):
    test_data.append(test)
    all_recs.append(eval('ItemItem', algo_ii, train, test))
    all_recs.append(eval('ALS', algo_als, train, test))
```

With the results in place, we can concatenate them into a single data frame:

[8]:
```
all_recs = pd.concat(all_recs, ignore_index=True)
all_recs.head()
```

[8]:
```
   item     score  user  rank  rating Algorithm
0   603  4.742555     6     1     0.0  ItemItem
1   357  4.556866     6     2     4.0  ItemItem
2  1398  4.493086     6     3     0.0  ItemItem
3   611  4.477099     6     4     0.0  ItemItem
4  1449  4.454879     6     5     0.0  ItemItem
```

[9]:
```
test_data = pd.concat(test_data, ignore_index=True)
```

nDCG is a per-user metric. Let's compute it for each user.

However, there is a little nuance: the recommendation list does not contain the information needed to normalize the DCG. Specifically, the nDCG depends on *all* the user's test items.

So we need to do three things:

1. Compute DCG of the recommendation lists.
2. Compute ideal DCGs for each test user
3. Combine and compute normalized versions

We do assume here that each user only appears once per algorithm. Since our crossfold method partitions users, this is fine.

[10]:
```
user_dcg = all_recs.groupby(['Algorithm', 'user']).rating.apply(tnmetrics.dcg)
user_dcg = user_dcg.reset_index(name='DCG')
user_dcg.head()
```

[10]:
```
  Algorithm  user        DCG
0       ALS     1  11.556574
1       ALS     2   7.383188
2       ALS     3   1.223253
3       ALS     4   0.000000
4       ALS     5   4.857249
```

[11]:
```
ideal_dcg = tnmetrics.compute_ideal_dcgs(test)
ideal_dcg.head()
```

[11]:
```
   user  ideal_dcg
0     4  16.946678
1    14  34.937142
2    15  25.770188
3    22  34.698538
4    23  41.289861
```

[12]:
```
user_ndcg = pd.merge(user_dcg, ideal_dcg)
user_ndcg['nDCG'] = user_ndcg.DCG / user_ndcg.ideal_dcg
user_ndcg.head()
```

[12]:
```
  Algorithm  user       DCG  ideal_dcg      nDCG
0       ALS     4  0.000000  16.946678  0.000000
1  ItemItem     4  0.000000  16.946678  0.000000
2       ALS    14  7.060065  34.937142  0.202079
3  ItemItem    14  7.218123  34.937142  0.206603
4       ALS    15  1.773982  25.770188  0.068839
```

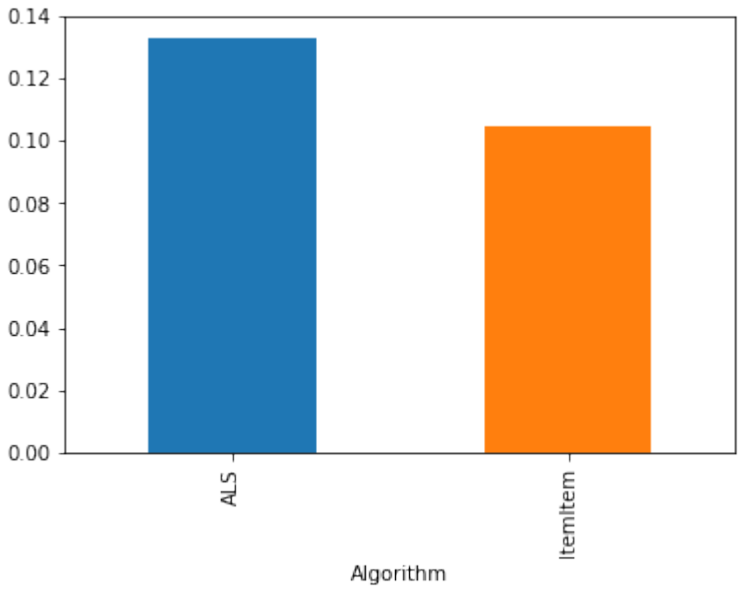
Now we have nDCG values!

[13]:
```
user_ndcg.groupby('Algorithm').nDCG.mean()
```

[13]:
```
Algorithm
ALS         0.133029
ItemItem    0.104659
Name: nDCG, dtype: float64
```

[14]:
```
user_ndcg.groupby('Algorithm').nDCG.mean().plot.bar()
```

[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2643b7a0cf8>
```




[ ]:

## 2.2 Crossfold preparation

The LKPY *crossfold* module provides support for preparing data sets for cross-validation. Crossfold methods are implemented as functions that operate on data frames and return generators of *(train, test)* pairs (*lenskit.crossfold.TTPair* objects). The train and test objects in each pair are also data frames, suitable for evaluation or writing out to a file.

Crossfold methods make minimal assumptions about their input data frames, so the frames can be ratings, purchases, or whatever. They do assume that each row represents a single data point for the purpose of splitting and sampling.

Experiment code should generally use these functions to prepare train-test files for training and evaluating algorithms. For example, the following will perform a user-based 5-fold cross-validation as was the default in the old LensKit:

```
import pandas as pd
import lenskit.crossfold as xf
ratings = pd.read_csv('ml-20m/ratings.csv')
ratings = ratings.rename(columns={'userId': 'user', 'movieId': 'item'})
for i, tp in enumerate(xf.partition_users(ratings, 5, xf.SampleN(5))):
    tp.train.to_csv('ml-20m.exp/train-%d.csv' % (i,))
    tp.train.to_parquet('ml-20m.exp/train-%d.parquet % (i,))
```

(continues on next page)

(continued from previous page)

```
    tp.test.to_csv('ml-20m.exp/test-%d.csv' % (i,))
    tp.test.to_parquet('ml-20m.exp/test-%d.parquet % (i,))
```

### 2.2.1 Row-based splitting

The simplest preparation methods sample or partition the rows in the input frame. A 5-fold **:py:funcc:'partition_rows'** split will result in 5 splits, each of which extracts 20% of the rows for testing and leaves 80% for training.

lenskit.crossfold.**partition_rows**(*data*, *partitions*)

Partition a frame of ratings or other datainto train-test partitions. This function does not care what kind of data is in *data*, so long as it is a Pandas DataFrame (or equivalent).

**Parameters**

- **data** (pandas.DataFrame or equivalent) – a data frame containing ratings or other data you wish to partition.
- **partitions** (*integer*) – the number of partitions to produce

**Return type** iterator

**Returns** an iterator of train-test pairs

lenskit.crossfold.**sample_rows**(*data*, *partitions*, *size*, *disjoint=True*)

Sample train-test a frame of ratings into train-test partitions. This function does not care what kind of data is in *data*, so long as it is a Pandas DataFrame (or equivalent).

**Parameters**

- **data** (pandas.DataFrame or equivalent) – a data frame containing ratings or other data you wish to partition.
- **partitions** (*integer*) – the number of partitions to produce

**Return type** iterator

**Returns** an iterator of train-test pairs

### 2.2.2 User-based splitting

It's often desirable to use users, instead of raw rows, as the basis for splitting data. This allows you to control the experimental conditions on a user-by-user basis, e.g. by making sure each user is tested with the same number of ratings. These methods require that the input data frame have a *user* column with the user names or identifiers.

The algorithm used by each is as follows:

1. Sample or partition the set of user IDs into *n* sets of test users.
2. For each set of test users, select a set of that user's rows to be test rows.
3. **Create a training set for each test set consisting of the non-selected rows from each** of that set's test users, along with all rows from each non-test user.

lenskit.crossfold.**partition_users**(*data*, *partitions: int*, *method: lenskit.crossfold.PartitionMethod*)

Partition a frame of ratings or other data into train-test partitions user-by-user. This function does not care what kind of data is in *data*, so long as it is a Pandas DataFrame (or equivalent) and has a *user* column.

**Parameters**

- **data** (pandas.DataFrame or equivalent) – a data frame containing ratings or other data you wish to partition.
- **partitions** (*integer*) – the number of partitions to produce
- **method** – The method for selecting test rows for each user.

**Return type** iterator

**Returns** an iterator of train-test pairs

lenskit.crossfold.**sample_users**(*data*, *partitions: int*, *size: int*, *method: lenskit.crossfold.PartitionMethod*, *disjoint=True*)

Create train-test partitions by sampling users. This function does not care what kind of data is in *data*, so long as it is a Pandas DataFrame (or equivalent) and has a *user* column.

**Parameters**

- **data** (pandas.DataFrame or equivalent) – a data frame containing ratings or other data you wish to partition.
- **partitions** – the number of partitions to produce
- **size** – the sample size
- **method** – The method for selecting test rows for each user.
- **disjoint** – whether user samples should be disjoint

**Return type** iterator

**Returns** an iterator of train-test pairs

### Selecting user test rows

These functions each take a *method* to decide how select each user's test rows. The method is a function that takes a data frame (containing just the user's rows) and returns the test rows. This function is expected to preserve the index of the input data frame (which happens by default with common means of implementing samples).

We provide several partition method factories:

lenskit.crossfold.**SampleN**(*n*)

Randomly select a fixed number of test rows per user/item.

**Parameters** **n** – The number of test items to select.

lenskit.crossfold.**SampleFrac**(*frac*)

Randomly select a fraction of test rows per user/item.

**Parameters** **frac** – the fraction of items to select for testing.

lenskit.crossfold.**LastN**(*n*, *col='timestamp'*)

Select a fixed number of test rows per user/item, based on ordering by a column.

**Parameters**

- **n** – The number of test items to select.
- **col** – The column to sort by.

lenskit.crossfold.**LastFrac**(*frac*, *col='timestamp'*)

Select a fraction of test rows per user/item.

**Parameters**

- **frac** – the fraction of items to select for testing.

- **col** – The column to sort by.

### 2.2.3 Utility Classes

**class** lenskit.crossfold.**PartitionMethod**
Partition methods select test rows for a user or item. Partition methods are callable; when called with a data frame, they return the test rows.

**__call__**(*udf*)
Subset a data frame.

**Parameters** **udf** – The input data frame of rows for a user or item.

**Returns** The data frame of test rows, a subset of *udf*.

**class** lenskit.crossfold.**TTPair**
Train-test pair (named tuple).

**test**
Test data for this pair.

**train**
Train data for this pair.

## 2.3 Batch-Running Recommenders

The functions in *lenskit.batch* enable you to generate many recommendations or predictions at the same time, useful for evaluations and experiments.

### 2.3.1 Recommendation

lenskit.batch.**recommend**(*algo*, *users*, *n*, *candidates*, *ratings=None*, *nprocs=None*)
Batch-recommend for multiple users. The provided algorithm should be a `algorithms.Recommender` or `algorithms.Predictor` (which will be converted to a top-N recommender).

**Parameters**

- **algo** – the algorithm
- **model** – The algorithm model
- **users** (*array-like*) – the users to recommend for
- **n** (*int*) – the number of recommendations to generate (None for unlimited)
- **candidates** – the users' candidate sets. This can be a function, in which case it will be passed each user ID; it can also be a dictionary, in which case user IDs will be looked up in it.
- **ratings** (*pandas.DataFrame*) – if not `None`, a data frame of ratings to attach to recommendations when available.

**Returns** A frame with at least the columns `user`, `rank`, and `item`; possibly also `score`, and any other columns returned by the recommender.

### 2.3.2 Rating Prediction

lenskit.batch.**predict**(*algo*, *pairs*, *nprocs=None*)

Generate predictions for user-item pairs. The provided algorithm should be a `algorithms.Predictor` or a function of two arguments: the user ID and a list of item IDs. It should return a dictionary or a `pandas.Series` mapping item IDs to predictions.

**Parameters**

- **algo** (`lenskit.algorithms.Predictor`) – A rating predictor function or algorithm.
- **pairs** (*pandas.DataFrame*) – A data frame of (`user`, `item`) pairs to predict for. If this frame also contains a `rating` column, it will be included in the result.
- **nprocs** (*int*) – The number of processes to use for parallel batch prediction.

**Returns** a frame with columns `user`, `item`, and `prediction` containing the prediction results. If `pairs` contains a *rating* column, this result will also contain a *rating* column.

**Return type** pandas.DataFrame

### 2.3.3 Scripting Evaluation

**class** lenskit.batch.**MultiEval**(*path*, *predict=True*, *recommend=100*, *candidates=<class 'lenskit.topn.UnratedCandidates'>*, *nprocs=None*, *combine=True*)

A runner for carrying out multiple evaluations, such as parameter sweeps.

**Parameters**

- **path** (str or `pathlib.Path`) – the working directory for this evaluation. It will be created if it does not exist.
- **predict** (*bool*) – whether to generate rating predictions.
- **recommend** (*int*) – the number of recommendations to generate per user (None to disable top-N).
- **candidates** (*function*) – the default candidate set generator for recommendations. It should take the training data and return a candidate generator, itself a function mapping user IDs to candidate sets.
- **combine** (*bool*) – whether to combine output; if `False`, output will be left in separate files, if `True`, it will be in a single set of files (runs, recommendations, and preditions).

**add_algorithms**(*algos*, *parallel=False*, *attrs=[]*, ***kwargs*)

Add one or more algorithms to the run.

**Parameters**

- **algos** (*algorithm or list*) – the algorithm(s) to add.
- **parallel** (*bool*) – if `True`, allow this algorithm to be trained in parallel with others.
- **attrs** (*list of str*) – a list of attributes to extract from the algorithm objects and include in the run descriptions.
- **kwargs** – additional attributes to include in the run descriptions.

**add_datasets**(*data*, *name=None*, *candidates=None*, ***kwargs*)

Add one or more datasets to the run.

**Parameters**

- **data** – The input data set(s) to run. Can be one of the following:
  - A tuple of (train, test) data.
  - An iterable of (train, test) pairs, in which case the iterable is not consumed until it is needed.
  - A function yielding either of the above, to defer data load until it is needed.

  Data can be either data frames or paths; paths are loaded after detection using `util.read_df_detect()`.
- **kwargs** – additional attributes pertaining to these data sets.

**collect_results**()
Collect the results from non-combined runs into combined output files.

**persist_data**()
Persist the data for an experiment, replacing in-memory data sets with file names. Once this has been called, the sweep can be pickled.

**run**(*runs=None*)
Run the evaluation.

**Parameters** **runs** (*int or set-like*) – If provided, a specific set of runs to run. Useful for splitting an experiment into individual runs. This is a set of 1-based run IDs, not 0-based indexes.

**run_count**()
Get the number of runs in this evaluation.

## 2.4 Evaluating Recommender Output

LensKit's evaluation support is based on post-processing the output of recommenders and predictors. The batch utilities provide support for generating these outputs.

We generally recommend using Jupyter notebooks for evaluation.

### 2.4.1 Prediction Accuracy Metrics

The *lenskit.metrics.predict* module containins prediction accuracy metrics.

#### Metric Functions

lenskit.metrics.predict.**rmse**(*predictions*, *truth*, *missing='error'*)
Compute RMSE (root mean squared error).

**Parameters**

- **predictions** (*pandas.Series*) – the predictions
- **truth** (*pandas.Series*) – the ground truth ratings from data
- **missing** (*string*) – how to handle predictions without truth. Can be one of `'error'` or `'ignore'`.

**Returns** the root mean squared approximation error

**Return type** double

lenskit.metrics.predict.**mae**(*predictions*, *truth*, *missing='error'*)
Compute MAE (mean absolute error).

**Parameters**

- **predictions** (*pandas.Series*) – the predictions
- **truth** (*pandas.Series*) – the ground truth ratings from data
- **missing** (*string*) – how to handle predictions without truth. Can be one of `'error'` or `'ignore'`.

**Returns** the mean absolute approximation error

**Return type** double

#### Working with Missing Data

LensKit rating predictors do not report predictions when their core model is unable to predict. For example, a nearest-neighbor recommender will not score an item if it cannot find any suitable neighbors. Following the Pandas convention, these items are given a score of NaN (when Pandas implements better missing data handling, it will use that, so use `pandas.Series.isna()`/`pandas.Series.notna()`, not the `isnan` versions.

However, this causes problems when computing predictive accuracy: recommenders are not being tested on the same set of items. If a recommender only scores the easy items, for example, it could do much better than a recommender that is willing to attempt more difficult items.

A good solution to this is to use a *fallback predictor* so that every item has a prediction. In LensKit, *lenskit.algorithms.basic.Fallback* implements this functionality; it wraps a sequence of recommenders, and for each item, uses the first one that generates a score.

You set it up like this:

```
cf = ItemItem(20)
base = Bias(damping=5)
algo = Fallback(cf, base)
```

### 2.4.2 Top-*N* Accuracy Metrics

The *lenskit.metrics.topn* module contains metrics for evaluating top-*N* recommendation lists.

#### Classification Metrics

These metrics treat the recommendation list as a classification of relevant items.

lenskit.metrics.topn.**precision**(*recs*, *relevant*)
Compute the precision of a set of recommendations.

**Parameters**

- **recs** (*array-like*) – a sequence of recommended items
- **relevant** (*set-like*) – the set of relevant items

**Returns** the fraction of recommended items that are relevant

**Return type** double

lenskit.metrics.topn.**recall**(*recs*, *relevant*)
Compute the recall of a set of recommendations.

**Parameters**

- **recs** (*array-like*) – a sequence of recommended items
- **relevant** (*set-like*) – the set of relevant items

**Returns** the fraction of relevant items that were recommended.

**Return type** double

### Ranked List Metrics

These metrics treat the recommendation list as a ranked list of items that may or may not be relevant.

lenskit.metrics.topn.**recip_rank**(*recs*, *relevant*)
Compute the reciprocal rank of the first relevant item in a recommendation list. This is used to compute MRR.

**Parameters**

- **recs** (*array-like*) – a sequence of recommended items
- **relevant** (*set-like*) – the set of relevant items

**Returns** the reciprocal rank of the first relevant item.

**Return type** double

### Utility Metrics

The DCG function estimates a utility score for a ranked list of recommendations. The results can be combined with ideal DCGs to compute nDCG.

lenskit.metrics.topn.**dcg**(*scores*, *discount=<ufunc 'log2'>*)
Compute the Discounted Cumulative Gain of a series of recommended items with rating scores. These should be relevance scores; they can be $0, 1$ for binary relevance data.

Discounted cumultative gain is computed as:

$$\mathrm{DCG}(L, u) = \sum_{i=1}^{|L|} \frac{r_{ui}}{d(i)}$$

You will usually want *normalized* discounted cumulative gain; this is

$$\mathrm{nDCG}(L, u) = \frac{\mathrm{DCG}(L, u)}{\mathrm{DCG}(L_{\mathrm{ideal}}, u)}$$

Compute that by computing the DCG of the recommendations & the test data, then merge the results and divide. The *compute_ideal_dcgs()* function is helpful for preparing that data.

**Parameters**

- **scores** (*array-like*) – The utility scores of a list of recommendations, in recommendation order.
- **discount** (*ufunc*) – the rank discount function. Each item's score will be divided the discount of its rank, if the discount is greater than 1.

**Returns** the DCG of the scored items.

**Return type** double

lenskit.metrics.topn.**compute_ideal_dcgs**(*ratings*, *discount=<ufunc 'log2'>*)

Compute the ideal DCG for rating data. This groups the rating data by everything *except* its `item` and `rating` columns, sorts each group by rating, and computes the DCG.

**Parameters** **ratings** (*pandas.DataFrame*) – A rating data frame with `item`, `rating`, and other columns.

**Returns**

**The data frame of DCG values. The `item` and `rating` columns in `ratings` are replaced by an `ideal_dcg` column.**

**Return type** pandas.DataFrame

### 2.4.3 Loading Outputs

We typically store the output of recommendation runs in LensKit experiments in CSV or Parquet files. The *lenskit.batch.MultiEval* class arranges to run a set of algorithms over a set of data sets, and store the results in a collection of Parquet files in a specified output directory.

There are several files:

**runs.parquet** The _runs_, algorithm-dataset combinations. This file contains the names & any associated properties of each algorithm and data set run, such as a feature count.

**recommendations.parquet** The recommendations, with columns `RunId`, `user`, `rank`, `item`, and `rating`.

**predictions.parquet** The rating predictions, if the test data includes ratings.

For example, if you want to examine nDCG by neighborhood count for a set of runs on a single data set, you can do:

```
import pandas as pd
from lenskit.metrics import topn as lm

runs = pd.read_parquet('eval-dir/runs.parquet')
recs = pd.read_parquet('eval-dir/recs.parquet')
meta = runs.loc[:, ['RunId', 'max_neighbors']]

# compute each user's nDCG
user_ndcg = recs.groupby(['RunId', 'user']).rating.apply(lm.ndcg)
user_ndcg = user_ndcg.reset_index(name='nDCG')
# combine with metadata for feature count
user_ndcg = pd.merge(user_ndcg, meta)
# group and aggregate
nbr_ndcg = user_ndcg.groupby('max_neighbors').nDCG.mean()
nbr_ndcg.plot()
```

## 2.5 Algorithms

LKPY provides general algorithmic concepts, along with implementations of several algorithms. These algorithm interfaces are based on the SciKit design patterns [SKAPI], adapted for Pandas-based data structures.

### 2.5.1 Algorithm Interfaces

LKPY's batch routines and utility support for managing algorithms expect algorithms to implement consistent interfaces. This page describes those interfaces.

The interfaces are realized as abstract base classes with the Python `abc` module. Implementations must be registered with their interfaces, either by subclassing the interface or by calling `abc.ABCMeta.register()`.

#### Base Algorithm

Algorithms follow the SciKit fit-predict paradigm for estimators, except they know natively how to work with Pandas objects.

The *Algorithm* interface defines common methods.

**class** lenskit.algorithms.**Algorithm**

Base class for LensKit algorithms. These algorithms follow the SciKit design pattern for estimators.

**fit**(*ratings*, **args*, ***kwargs*)

Train a model using the specified ratings (or similar) data.

**Parameters**

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **args** – Additional training data the algorithm may require.
- **kwargs** – Additional training data the algorithm may require.

**Returns** The algorithm object.

**get_params**(*deep=True*)

Get the parameters for this algorithm (as in scikit-learn). Algorithm parameters should match constructor argument names.

The default implementation returns all attributes that match a constructor parameter name. It should be compatible with `scikit.base.BaseEstimator.get_params()` method so that LensKit alogrithms can be cloned with `scikit.base.clone()` as well as *lenskit.util.clone()*.

**Returns** the algorithm parameters.

**Return type** dict

**load**(*file*)

Load a fit algorithm from a file. The default implementation unpickles the object and transplants its parameters and model into this object.

**Parameters** **file** (*path-like*) – the file to load.

**save**(*file*)

Save a fit algorithm to a file. The default implementation pickles the object.

**Parameters** **file** (*path-like*) – the file to save.

#### Recommendation

The *Recommender* interface provides an interface to generating recommendations. Not all algorithms implement it; call `Recommender.adapt()` on an algorithm to get a recommender for any algorithm that at least implements *Predictor*. For example:

```
pred = Bias(damping=5)
rec = Recommender.adapt(pred)
```

**Note:** We are rethinking the ergonomics of this interface, and it may change in LensKit 0.6. We expect keep compatibility in the *lenskit.batch.recommend()* API, though.

**class** lenskit.algorithms.**Recommender**
Recommends lists of items for users.

**recommend**(*user*, *n=None*, *candidates=None*, *ratings=None*)
Compute recommendations for a user.

**Parameters**

- **user** – the user ID
- **n** (*int*) – the number of recommendations to produce (`None` for unlimited)
- **candidates** (*array-like*) – the set of valid candidate items.
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, they may be used to override or augment the model's notion of a user's preferences.

**Returns** a frame with an `item` column; if the recommender also produces scores, they will be in a `score` column.

**Return type** pandas.DataFrame

### Rating Prediction

**class** lenskit.algorithms.**Predictor**
Predicts user ratings of items. Predictions are really estimates of the user's like or dislike, and the `Predictor` interface makes no guarantees about their scale or granularity.

**predict**(*pairs*, *ratings=None*)
Compute predictions for user-item pairs. This method is designed to be compatible with the general SciKit paradigm; applications typically want to use *predict_for_user()*.

**Parameters**

- **pairs** (*pandas.DataFrame*) – The user-item pairs, as `user` and `item` columns.
- **ratings** (*pandas.DataFrame*) – user-item rating data to replace memorized data.

**Returns** The predicted scores for each user-item pair.

**Return type** pandas.Series

**predict_for_user**(*user*, *items*, *ratings=None*)
Compute predictions for a user and items.

**Parameters**

- **user** – the user ID
- **items** (*array-like*) – the items to predict
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, they may be used to override or augment the model's notion of a user's preferences.

**Returns** scores for the items, indexed by item id.

**Return type** pandas.Series

### 2.5.2 Basic and Utility Algorithms

The *lenskit.algorithms.basic* module contains baseline and utility algorithms for nonpersonalized recommendation and testing.

#### Personalized Mean Rating Prediction

**class** lenskit.algorithms.basic.**Bias**(*items=True*, *users=True*, *damping=0.0*)

Bases: *lenskit.algorithms.Predictor*

A user-item bias rating prediction algorithm. This implements the following predictor algorithm:

$$s(u,i) = \mu + b_i + b_u$$

where $\mu$ is the global mean rating, $b_i$ is item bias, and $b_u$ is the user bias. With the provided damping values $\beta_\mathrm{u}$ and $\beta_\mathrm{i}$, they are computed as follows:

$$\mu = \frac{\sum_{r_{ui} \in R} r_{ui}}{|R|} \qquad b_i = \frac{\sum_{r_{ui} \in R_i} (r_{ui} - \mu)}{|R_i| + \beta_\mathrm{i}} \qquad b_u = \frac{\sum_{r_{ui} \in R_u} (r_{ui} - \mu - b_i)}{|R_u| + \beta_\mathrm{u}}$$

The damping values can be interpreted as the number of default (mean) ratings to assume *a priori* for each user or item, damping low-information users and items towards a mean instead of permitting them to take on extreme values based on few ratings.

**Parameters**

- **items** – whether to compute item biases
- **users** – whether to compute user biases
- **damping** (*number or tuple*) – Bayesian damping to apply to computed biases. Either a number, to damp both user and item biases the same amount, or a (user,item) tuple providing separate damping values.

**mean_**
The global mean rating.

**Type** double

**item_offsets_**
The item offsets ($b_i$ values)

**Type** pandas.Series

**user_offsets_**
The item offsets ($b_u$ values)

**Type** pandas.Series

**fit**(*data*)
Train the bias model on some rating data.

**Parameters** **data** (*DataFrame*) – a data frame of ratings. Must have at least *user*, *item*, and *rating* columns.

**Returns** the fit bias object.

**Return type** *Bias*

**predict_for_user**(*user*, *items*, *ratings=None*)
Compute predictions for a user and items. Unknown users and items are assumed to have zero bias.

**Parameters**

- **user** – the user ID
- **items** (*array-like*) – the items to predict
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, will be used to recompute the user's bias at prediction time.

**Returns** scores for the items, indexed by item id.

**Return type** pandas.Series

### Fallback Predictor

The `Fallback` rating predictor is a simple hybrid that takes a list of composite algorithms, and uses the first one to return a result to predict the rating for each item.

A common case is to fill in with *Bias* when a primary predictor cannot score an item.

**class** lenskit.algorithms.basic.**Fallback**(*algorithms*, **others*)
Bases: *lenskit.algorithms.Predictor*

The Fallback algorithm predicts with its first component, uses the second to fill in missing values, and so forth.

**fit**(*ratings*, **args*, ***kwargs*)
Train a model using the specified ratings (or similar) data.

**Parameters**

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **args** – Additional training data the algorithm may require.
- **kwargs** – Additional training data the algorithm may require.

**Returns** The algorithm object.

**load**(*file*)
Load a fit algorithm from a file. The default implementation unpickles the object and transplants its parameters and model into this object.

**Parameters** **file** (*path-like*) – the file to load.

**predict_for_user**(*user*, *items*, *ratings=None*)
Compute predictions for a user and items.

**Parameters**

- **user** – the user ID
- **items** (*array-like*) – the items to predict
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, they may be used to override or augment the model's notion of a user's preferences.

**Returns** scores for the items, indexed by item id.

**Return type** pandas.Series

**save**(*path*)
Save a fit algorithm to a file. The default implementation pickles the object.

**Parameters** **file** (*path-like*) – the file to save.

#### Memorized Predictor

The `Memorized` recommender is primarily useful for test cases. It memorizes a set of rating predictions and returns them.

**class** lenskit.algorithms.basic.**Memorized**(*scores*)
Bases: *lenskit.algorithms.Predictor*

The memorized algorithm memorizes socres provided at construction time.

**fit**(**args*, ***kwargs*)
Train a model using the specified ratings (or similar) data.

**Parameters**

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **args** – Additional training data the algorithm may require.
- **kwargs** – Additional training data the algorithm may require.

**Returns** The algorithm object.

**predict_for_user**(*user*, *items*, *ratings=None*)
Compute predictions for a user and items.

**Parameters**

- **user** – the user ID
- **items** (*array-like*) – the items to predict
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, they may be used to override or augment the model's notion of a user's preferences.

**Returns** scores for the items, indexed by item id.

**Return type** pandas.Series

### 2.5.3 k-NN Collaborative Filtering

LKPY provides user- and item-based classical k-NN collaborative Filtering implementations. These lightly-configurable implementations are intended to capture the behavior of the Java-based LensKit implementations to provide a good upgrade path and enable basic experiments out of the box.

#### Item-based k-NN

**class** lenskit.algorithms.item_knn.**ItemItem**(*nnbrs*, *min_nbrs=1*, *min_sim=1e-06*, *save_nbrs=None*, *center=True*, *aggregate='weighted-average'*)
Bases: *lenskit.algorithms.Predictor*

Item-item nearest-neighbor collaborative filtering with ratings. This item-item implementation is not terribly configurable; it hard-codes design decisions found to work well in the previous Java-based LensKit code.

**item_index_**
the index of item IDs.

**Type** pandas.Index

**item_means_**
the mean rating for each known item.

**Type** numpy.ndarray

**item_counts_**
the number of saved neighbors for each item.

**Type** numpy.ndarray

**sim_matrix_**
the similarity matrix.

**Type** *matrix.CSR*

**user_index_**
the index of known user IDs for the rating matrix.

**Type** pandas.Index

**rating_matrix_**
the user-item rating matrix for looking up users' ratings.

**Type** *matrix.CSR*

**fit**(*ratings*)
Train a model.

The model-training process depends on `save_nbrs` and `min_sim`, but *not* on other algorithm parameters.

**Parameters** **ratings** (*pandas.DataFrame*) – (user,item,rating) data for computing item similarities.

**load**(*path*)
Load a fit algorithm from a file. The default implementation unpickles the object and transplants its parameters and model into this object.

**Parameters** **file** (*path-like*) – the file to load.

**predict_for_user**(*user*, *items*, *ratings=None*)
Compute predictions for a user and items.

**Parameters**

- **user** – the user ID
- **items** (*array-like*) – the items to predict
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, they may be used to override or augment the model's notion of a user's preferences.

**Returns** scores for the items, indexed by item id.

**Return type** pandas.Series

**save**(*path*)
Save a fit algorithm to a file. The default implementation pickles the object.

**Parameters** **file** (*path-like*) – the file to save.

### User-based k-NN

**class** lenskit.algorithms.user_knn.**UserUser**(*nnbrs*, *min_nbrs=1*, *min_sim=0*, *center=True*, *aggregate='weighted-average'*)

Bases: *lenskit.algorithms.Predictor*

User-user nearest-neighbor collaborative filtering with ratings. This user-user implementation is not terribly configurable; it hard-codes design decisions found to work well in the previous Java-based LensKit code.

**user_index_**
User index.

**Type** pandas.Index

**item_index_**
Item index.

**Type** pandas.Index

**user_means_**
User mean ratings.

**Type** numpy.ndarray

**rating_matrix_**
Normalized user-item rating matrix.

**Type** *matrix.CSR*

**transpose_matrix_**
Transposed un-normalized rating matrix.

**Type** *matrix.CSR*

**fit**(*ratings*)
"Train" a user-user CF model. This memorizes the rating data in a format that is usable for future computations.

**Parameters** **ratings** (*pandas.DataFrame*) – (user, item, rating) data for collaborative filtering.

**Returns** a memorized model for efficient user-based CF computation.

**Return type** UUModel

**load**(*path*)
Load a fit algorithm from a file. The default implementation unpickles the object and transplants its parameters and model into this object.

**Parameters** **file** (*path-like*) – the file to load.

**predict_for_user**(*user*, *items*, *ratings=None*)
Compute predictions for a user and items.

**Parameters**

- **user** – the user ID
- **items** (*array-like*) – the items to predict
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, will be used to recompute the user's bias at prediction time.

**Returns** scores for the items, indexed by item id.

**Return type** pandas.Series

**save**(*path*)
Save a fit algorithm to a file. The default implementation pickles the object.

**Parameters** **file** (*path-like*) – the file to save.

### 2.5.4 Classic Matrix Factorization

LKPY provides classical matrix factorization implementations.



#### Common Support

The `mf_common` module contains common support code for matrix factorization algorithms.

**class** lenskit.algorithms.mf_common.**MFPredictor**
Common predictor for matrix factorization.

**user_index_**
Users in the model (length=:math:$m$).

**Type** pandas.Index

**item_index_**
Items in the model (length=:math:$n$).

**Type** pandas.Index

**user_features_**
The $m \times k$ user-feature matrix.

**Type** numpy.ndarray

**item_features_**
The $n \times k$ item-feature matrix.

**Type** numpy.ndarray

**load**(*path*)
Load a fit algorithm from a file. The default implementation unpickles the object and transplants its parameters and model into this object.

**Parameters** **file** (*path-like*) – the file to load.

**lookup_items**(*items*)
Look up the indices for a set of items.

**Parameters** **items** (*array-like*) – the item IDs to look up.

**Returns** the item indices. Unknown items will have negative indices.

**Return type** numpy.ndarray

**lookup_user**(*user*)
Look up the index for a user.

**Parameters** **user** – the user ID to look up

**Returns** the user index.

**Return type** int

**n_features**
The number of features.

**n_items**
The number of items.

**n_users**
The number of users.

**save**(*path*)
Save a fit algorithm to a file. The default implementation pickles the object.

**Parameters** **file** (*path-like*) – the file to save.

**score**(*user*, *items*)
Score a set of items for a user. User and item parameters must be indices into the matrices.

**Parameters**

- **user** (*int*) – the user index
- **items** (*array-like of int*) – the item indices
- **raw** (*bool*) – if `True`, do return raw scores without biases added back.

**Returns** the scores for the items.

**Return type** numpy.ndarray

**class** lenskit.algorithms.mf_common.**BiasMFPredictor**
Common model for biased matrix factorization.

**user_index_**
Users in the model (length=:math:*m*).

**Type** pandas.Index

**item_index_**
Items in the model (length=:math:*n*).

**Type** pandas.Index

**global_bias_**
The global bias term.

**Type** double

**user_bias_**
The user bias terms.

**Type** numpy.ndarray

**item_bias_**
The item bias terms.

**Type** numpy.ndarray

**user_features_**
The $m \times k$ user-feature matrix.

**Type** numpy.ndarray

**item_features_**
The $n \times k$ item-feature matrix.

**Type** numpy.ndarray

**load**(*path: pathlib.Path*)
Load a fit algorithm from a file. The default implementation unpickles the object and transplants its parameters and model into this object.

**Parameters** **file** (*path-like*) – the file to load.

**save**(*path*)
Save a fit algorithm to a file. The default implementation pickles the object.

**Parameters** **file** (*path-like*) – the file to save.

**score**(*user*, *items*, *raw=False*)
Score a set of items for a user. User and item parameters must be indices into the matrices.

**Parameters**

- **user** (*int*) – the user index
- **items** (*array-like of int*) – the item indices
- **raw** (*bool*) – if `True`, do return raw scores without biases added back.

**Returns** the scores for the items.

**Return type** numpy.ndarray

### Alternating Least Squares

LensKit provides alternating least squares implementations of matrix factorization suitable for explicit feedback data. These implementations are parallelized with Numba, and perform best with the MKL from Conda.

### FunkSVD

FunkSVD is an SVD-like matrix factorization that uses stochastic gradient descent, configured much like coordinate descent, to train the user-feature and item-feature matrices.

**class** lenskit.algorithms.funksvd.**FunkSVD**(*features*, *iterations=100*, ***, *lrate=0.001*, *reg=0.015*, *damping=5*, *range=None*, *bias=True*)
Algorithm class implementing FunkSVD matrix factorization.

**Parameters**

- **features** (*int*) – the number of features to train
- **iterations** (*int*) – the number of iterations to train each feature
- **lrate** (*double*) – the learning rate
- **reg** (*double*) – the regularization factor
- **damping** (*double*) – damping factor for the underlying mean
- **bias** (Predictor) – the underlying bias model to fit. If `True`, then a *basic.Bias* model is fit with `damping`.
- **range** (*tuple*) – the `(min, max)` rating values to clamp ratings, or `None` to leave predictions unclamped.

**fit** (*ratings*)
Train a FunkSVD model.

**Parameters** **ratings** – the ratings data frame.

**predict_for_user** (*user*, *items*, *ratings=None*)
Compute predictions for a user and items.

**Parameters**

- **user** – the user ID
- **items** (*array-like*) – the items to predict
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, they may be used to override or augment the model's notion of a user's preferences.

**Returns** scores for the items, indexed by item id.

**Return type** pandas.Series

### 2.5.5 Hierarchical Poisson Factorization

This module provides a LensKit bridge to the hpfrec library implementing hierarchical Poisson factorization [GHB2013].

**class** lenskit.algorithms.hpf.**HPF** (*features*, ***kwargs*)
Hierarchical Poisson factorization, provided by hpfrec.

**Parameters**

- **features** (*int*) – the number of features
- ****kwargs** – arguments passed to hpfrec.HPF.

**fit** (*ratings*)
Train a model using the specified ratings (or similar) data.

**Parameters**

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **args** – Additional training data the algorithm may require.
- **kwargs** – Additional training data the algorithm may require.

**Returns** The algorithm object.

**predict_for_user** (*user*, *items*, *ratings=None*)
Compute predictions for a user and items.

**Parameters**

- **user** – the user ID
- **items** (*array-like*) – the items to predict
- **ratings** (*pandas.Series*) – the user's ratings (indexed by item id); if provided, they may be used to override or augment the model's notion of a user's preferences.

**Returns** scores for the items, indexed by item id.

**Return type** pandas.Series

### 2.5.6 Implicit

This module provides a LensKit bridge to the implicit library implementing several implicit-feedback recommenders.

**class** lenskit.algorithms.implicit.**ALS**(**args*, ***kwargs*)
LensKit interface to `implicit.als`.

**class** lenskit.algorithms.implicit.**BPR**(**args*, ***kwargs*)
LensKit interface to `implicit.bpr`.

## 2.6 Utility Functions

### 2.6.1 Matrix Utilities

We have some matrix-related utilities, since matrices are used so heavily in recommendation algorithms.

#### Building Ratings Matrices

lenskit.matrix.**sparse_ratings**(*ratings*, *scipy=False*)
Convert a rating table to a sparse matrix of ratings.

**Parameters**

- **ratings** (*pandas.DataFrame*) – a data table of (user, item, rating) triples.
- **scipy** – if `True`, return a SciPy matrix instead of *CSR*.

**Returns** a named tuple containing the sparse matrix, user index, and item index.

**Return type** *RatingMatrix*

**class** lenskit.matrix.**RatingMatrix**
A rating matrix with associated indices.

**matrix**
The rating matrix, with users on rows and items on columns.

**Type** *CSR* or scipy.sparse.csr_matrix

**users**
mapping from user IDs to row numbers.

**Type** pandas.Index

**items**
mapping from item IDs to column numbers.

**Type** pandas.Index

#### Compressed Sparse Row Matrices

We use CSR-format sparse matrices in quite a few places. Since SciPy's sparse matrices are not directly usable from Numba, we have implemented a Numba-compiled CSR representation that can be used from accelerated algorithm implementations.

lenskit.matrix.**csr_from_coo**(*rows*, *cols*, *vals*, *shape=None*)
Create a CSR matrix from data in COO format.

**Parameters**

- **rows** (*array-like*) – the row indices.
- **cols** (*array-like*) – the column indices.
- **vals** (*array-like*) – the data values; can be `None`.
- **shape** (*tuple*) – the array shape, or `None` to infer from row & column indices.

`lenskit.matrix.`**csr_from_scipy**(*mat*, *copy=True*)

Convert a scipy sparse matrix to an internal CSR.

**Parameters**

- **mat** (*scipy.sparse.spmatrix*) – a SciPy sparse matrix.
- **copy** (*bool*) – if `False`, reuse the SciPy storage if possible.

**Returns** a CSR matrix.

**Return type** *CSR*

`lenskit.matrix.`**csr_to_scipy**(*mat*)

Convert a CSR matrix to a SciPy `scipy.sparse.csr_matrix`.

**Parameters** **mat** (`CSR`) – A CSR matrix.

**Returns** A SciPy sparse matrix with the same data. It shares storage with `matrix`.

**Return type** scipy.sparse.csr_matrix

`lenskit.matrix.`**csr_rowinds**(*csr*)

Get the row indices for a CSR matrix.

**Parameters** **csr** (`CSR`) – a CSR matrix.

**Returns** the row index array for the CSR matrix.

**Return type** np.ndarray

`lenskit.matrix.`**csr_save**(*csr: numba.jitclass.base.CSR*, *prefix=None*)

Extract data needed to save a CSR matrix. This is intended to be used with, for example, `numpy.savez()` to save a matrix:

```
np.savez_compressed('file.npz', **csr_save(csr))
```

The `prefix` allows multiple matrices to be saved in a single file:

```
data = {}
data.update(csr_save(m1, prefix='m1'))
data.update(csr_save(m2, prefix='m2'))
np.savez_compressed('file.npz', **data)
```

**Parameters**

- **csr** (`CSR`) – the matrix to save.
- **prefix** (*str*) – the prefix for the data keys.

**Returns** a dictionary of data to save the matrix.

**Return type** dict

`lenskit.matrix.`**csr_load**(*data*, *prefix=None*)

Rematerialize a CSR matrix from loaded data. The inverse of *csr_save()*.

**Parameters**

- **data** (*dict-like*) – the input data.
- **prefix** (*str*) – the prefix for the data keys.

**Returns** the matrix described by `data`.

**Return type** *CSR*

**class** lenskit.matrix.**CSR**(*nrows*, *ncols*, *nnz*, *ptrs*, *inds*, *vals*)

Simple compressed sparse row matrix. This is like `scipy.sparse.csr_matrix`, with a couple of useful differences:

- It is a Numba jitclass, so it can be directly used from Numba-optimized functions.
- The value array is optional, for cases in which only the matrix structure is required.
- The value array, if present, is always double-precision.

You generally don't want to create this class yourself. Instead, use one of the related utility functions.

**nrows**
the number of rows.

**Type** int

**ncols**
the number of columns.

**Type** int

**nnz**
the number of entries.

**Type** int

**rowptrs**
the row pointers.

**Type** numpy.ndarray

**colinds**
the column indices.

**Type** numpy.ndarray

**values**
the values

**Type** numpy.ndarray

## 2.6.2 Math utilities

### Solvers

## 2.6.3 Miscellaneous

Miscellaneous utility functions.

lenskit.util.**clone**(*algo*)

Clone an algorithm, but not its fitted data. This is like `scikit.base.clone()`, but may not work on arbitrary SciKit estimators. LensKit algorithms are compatible with SciKit clone, however, so feel free to use that if you need more general capabilities.

This function is somewhat derived from the SciKit one.

```
>>> from lenskit.algorithms.basic import Bias
>>> orig = Bias()
>>> copy = clone(orig)
>>> copy is orig
False
>>> copy.damping == orig.damping
True
```

lenskit.util.**fspath**(*path*)
Backport of `os.fspath()` function for Python 3.5.

lenskit.util.**read_df_detect**(*path*)
Read a Pandas data frame, auto-detecting the file format based on filename suffix. The following file types are supported:

**CSV** File has suffix `.csv`, read with `pandas.read_csv()`.

**Parquet** File has suffix `.parquet`, `.parq`, or `.pq`, read with `pandas.read_parquet()`.

lenskit.util.**write_parquet**(*path*, *frame*, *append=False*)
Write a Parquet file.

**Parameters**

- **path** (*pathlib.Path*) – The path of the Parquet file to write.
- **frame** (*pandas.DataFrame*) – The data to write.
- **append** (*bool*) – Whether to append to the file or overwrite it.

## 2.7 Release Notes

### 2.7.1 0.5.0

LensKit 0.5.0 modifies the algorithm APIs to follow the SciKit design patterns instead of our previous custom patterns. Highlights of this change:

- Algorithms are trained in-place — we no longer have distinct model objects.
- Model data is stored as attributes on the algorithm object that end in _.
- Instead of writing `model = algo.train_model(ratings)`, call `algo.fit(ratings)`.

We also have some new capabilities:

- Ben Frederickson's Implicit library

### 2.7.2 0.3.0

A number of improvements, including replacing Cython/OpenMP with Numba and adding ALS.

### 2.7.3 0.2.0

A lot of fixes to get ready for RecSys.

### 2.7.4 0.1.0

Hello, world!

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

# Bibliography

[SKAPI] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake Vanderplas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. arXiv:1309.0238 [cs.LG].

[GHB2013] Prem Gopalan, Jake M. Hofman, and David M. Blei. 2013. Scalable Recommendation with Poisson Factorization. arXiv:1311.1704 [cs, stat] (November 2013). Retrieved February 9, 2017 from http://arxiv.org/abs/1311.1704.

# Python Module Index

## l

# Index

## Symbols



## A



## B



## C



## D



## F



## G



## H



## I

## L



## M



## N



## P



## R

## S



## T



## U



## V



## W