LensKit Documentation

Release 0.10.0

Michael D. Ekstrand

Jun 09, 2020

OVERVIEW

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 toolkit and a part of the LensKit project.

If you use Lenskit in published research, cite [LKPY].

CHAPTER

ONE

RESOURCES

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

1.1 Install LensKit

To install the current release with Anaconda (recommended):

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

The packages in the `lenskit` channel are intended to be used with Anaconda's default channels. We publish packages for Python 3.6, 3.7, and 3.8.

You can also use `pip` to install LensKit in a stock Python environment, such as a virtual environment:

```
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.

Note: LensKit is optimized for MKL-based Anaconda installs. It works in other Python environments, but performance will usually suffer for some algorithms. *lenskit.algorithms.item_knn* is particularly affected by this.

1.2 Getting Started

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

This notebook is also available on Google Collaboratory and nbviewer.

1.2.1 Setup

We first import the LensKit components we need:

[1]:
```
from lenskit.datasets import ML100K
from lenskit import batch, topn, util
from lenskit import crossfold as xf
from lenskit.algorithms import Recommender, als, item_knn as knn
from lenskit import topn
```

And Pandas is very useful:

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

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

1.2.2 Loading Data

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

[4]:
```
ml100k = ML100K('ml-100k')
ratings = ml100k.ratings
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

1.2.3 Defining Algorithms

Let's set up two algorithms:

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

1.2.4 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.

Note 2: our algorithms do not necessarily implement the `Recommender` interface, so we adapt them. This fills in a default candidate selector.

The code function looks like this:

[6]:
```
def eval(aname, algo, train, test):
    fittable = util.clone(algo)
    fittable = Recommender.adapt(fittable)
    fittable.fit(train)
    users = test.user.unique()
    # now we run the recommender
    recs = batch.recommend(fittable, users, 100)
    # add the algorithm name for analyzability
    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	Algorithm
0	285	4.543364	5	1	ItemItem
1	1449	4.532999	5	2	ItemItem
2	1251	4.494639	5	3	ItemItem
3	114	4.479512	5	4	ItemItem
4	166	4.399639	5	5	ItemItem

To compute our analysis, we also need to concatenate the test data into a single frame:

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

We analyze our recommendation lists with a `RecListAnalysis`. It takes care of the hard work of making sure that the truth data (our test data) and the recoommendations line up properly.

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

[10]:
```
rla = topn.RecListAnalysis()
rla.add_metric(topn.ndcg)
results = rla.compute(all_recs, test_data)
results.head()
```

```
/home/MICHAELEKSTRAND/anaconda3/envs/lkpy-dev/lib/python3.7/site-packages/pandas/core/
↪indexing.py:1494: PerformanceWarning: indexing past lexsort depth may impact␣
↪performance.
  return self._getitem_tuple(key)
```

[10]:

		ndcg
user	Algorithm	
1	ALS	0.265268
	ItemItem	0.259708

(continues on next page)

(continued from previous page)

```
2    ALS         0.148335
     ItemItem    0.081890
3    ALS         0.026615
```

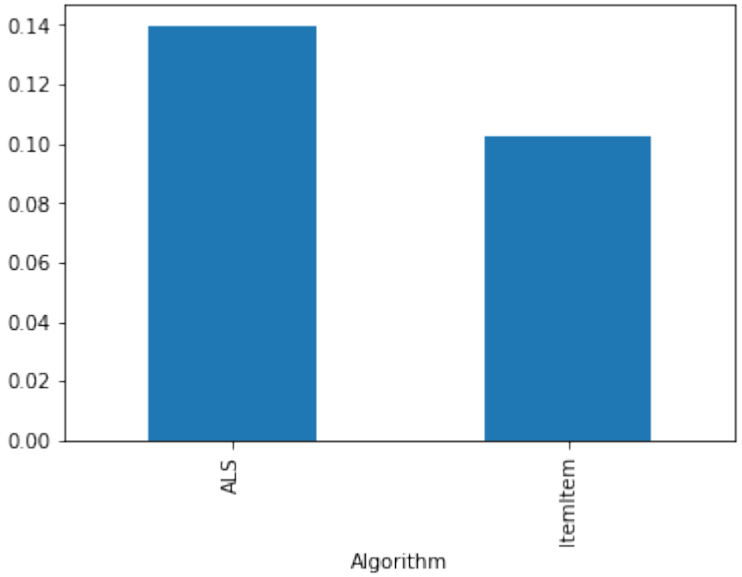
Now we have nDCG values!

[11]:
```
results.groupby('Algorithm').ndcg.mean()
```

[11]:
```
Algorithm
ALS         0.139689
ItemItem    0.102075
Name: ndcg, dtype: float64
```

[12]:
```
results.groupby('Algorithm').ndcg.mean().plot.bar()
```

[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f03842f8860>
```



[ ]:
```
```

1.3 Examples

There are several examples to help you see LensKit in action:

- The Getting Started guide
- The LensKit demo experiment
- Our extended Book Gender paper shows LensKit in use in an advanced experiment

In addition, PBS's Crash Course AI episode on recommender systems demonstrates LensKit in the video and associated Collaboratory notebook.

1.4 Loading Data

LensKit can work with any data in a `pandas.DataFrame` with the expected columns. LensKit algorithms expect a `ratings` frame to contain the following columns (in any order):

- `user`, containing user identifiers. No requirements are placed on user IDs — if an algorithm requires something specific, such as contiguous 0-based identifiers for indexing into an array — it will use a `pandas.Index` to map them.
- `item`, containing item identifiers. The same comments apply as for `user`.
- `rating`, containing user ratings (if available). Implicit-feedback code will not require ratings.

'Rating' data can contain other columns as well, and is a catch-all for any user-item interaction data. Algorithms will document any non-standard columns they can make use of.

`lenskit.algorithms.Recommender.fit()` can also accept additional data objects as keyword arguments, and algorithms that wrap other algorithms will pass this data through unchanged. Algorithms ignore extra data objects they receive. This allows you to build algorithms that train on data besides user-item interactions, such as user metadata or item content.

1.4.1 Data Loaders

The *`lenskit.datasets`* module provides utilities for reading a variety of commonly-used LensKit data sets. It does not package or automatically download them, but loads them from a local directory where you have unpacked the data set. Each data set class or function takes a `path` parameter specifying the location of the data set.

The normal mode of operation for these utilities is to provide a class for the data set; this class then exposes the data set's data as attributes. These attributes are cached internally, so e.g. accessing *`MovieLens.ratings`* twice will only load the data file once.

These data files have normalized column names to fit with LensKit's general conventions. These are the following:

- User ID columns are called `user`.
- Item ID columns are called `item`.
- Rating columns are called `rating`.
- Timestamp columns are called `timestamp`.

Other column names are unchanged. Data tables that provide information about specific things, such as a table of movie titles, are indexed by the relevant ID (e.g. *`MovieLens.ratings`* is indexed by `item`).

Data sets supported:

- *`MovieLens`*
- *`ML100K`*
- *`ML1M`*
- *`ML10M`*

1.4.2 MovieLens Data Sets

The GroupLens research group provides several data sets extracted from the MovieLens service [HK2015]. These can be downloaded from https://grouplens.org/datasets/movielens/.

class lenskit.datasets.**MovieLens**(*path='data/ml-20m'*)

Bases: object

Code for reading current MovieLens data sets, including ML-20M, ML-Latest, and ML-Latest-Small.

Parameters **path** (*str or pathlib.Path*) – Path to the directory containing the data set.

property ratings

The rating table.

```
>>> mlsmall = MovieLens('data/ml-latest-small')
>>> mlsmall.ratings
        user  item  rating   timestamp
0          1    31     2.5  1260759144
1          1  1029     3.0  1260759179
2          1  1061     3.0  1260759182
3          1  1129     2.0  1260759185
4          1  1172     4.0  1260759205
...
[100004 rows x 4 columns]
```

property movies

The movie table, with titles and genres. It is indexed by movie ID.

```
>>> mlsmall = MovieLens('data/ml-latest-small')
>>> mlsmall.movies
                                                    title
↪                       genres
item
1                                          Toy Story (1995)
↪Adventure|Animation|Children|Comedy|Fantasy
2                                            Jumanji (1995)
↪  Adventure|Children|Fantasy
3                                   Grumpier Old Men (1995)
↪               Comedy|Romance
4                                  Waiting to Exhale (1995)
↪        Comedy|Drama|Romance
5                        Father of the Bride Part II (1995)
↪                        Comedy
...
[9125 rows x 2 columns]
```

property links

The movie link table, connecting movie IDs to external identifiers. It is indexed by movie ID.

```
>>> mlsmall = MovieLens('data/ml-latest-small')
>>> mlsmall.links
         imdbId  tmdbId
item
1        114709     862
2        113497    8844
3        113228   15602
4        114885   31357
5        113041   11862
```

(continues on next page)

(continued from previous page)

```
...
[9125 rows x 2 columns]
```

property tags

The tag application table, recording user-supplied tags for movies.

```
>>> mlsmall = MovieLens('data/ml-latest-small')
>>> mlsmall.tags
      user  ...   timestamp
0       15  ...  1138537770
1       15  ...  1193435061
2       15  ...  1170560997
3       15  ...  1170626366
4       15  ...  1141391765
...
[1296 rows x 4 columns]
```

property tag_genome

The tag genome table, recording inferred item-tag relevance scores. This gets returned as a wide Pandas data frame, with rows indexed by item ID.

```
>>> ml20m = MovieLens('data/ml-20m')
>>> ml20m.tag_genome
tag          007  007 (series)  18th century  ...      wwii   zombie  zombies
item                                          ...
1        0.02500       0.02500       0.05775  ...   0.03625  0.07775  0.02300
2        0.03975       0.04375       0.03775  ...   0.01475  0.09025  0.01875
3        0.04350       0.05475       0.02800  ...   0.01950  0.09700  0.01850
4        0.03725       0.03950       0.03675  ...   0.01525  0.06450  0.01300
5        0.04200       0.05275       0.05925  ...   0.01675  0.10750  0.01825
...
[10381 rows x 1128 columns]
```

class lenskit.datasets.**ML100K**(*path='data/ml-100k'*)

Bases: object

The MovieLens 100K data set. This older data set is in a different format from the more current data sets loaded by *MovieLens*.

property available

Query whether the data set exists.

property ratings

Return the rating data (from `u.data`).

```
>>> ml = ML100K()
>>> ml.ratings
       user  item  rating  timestamp
0       196   242     3.0  881250949
1       186   302     3.0  891717742
2        22   377     1.0  878887116
3       244    51     2.0  880606923
4       166   346     1.0  886397596
...
[100000 rows x 4 columns]
```

property users

Return the user data (from `u.user`).

```
>>> ml = ML100K()
>>> ml.users
      age gender     occupation     zip
user
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
...
[943 rows x 4 columns]
```

property movies

Return the user data (from `u.user`).

```
>>> ml = ML100K()
>>> ml.movies
                                          title      release  ...  War Western
item                                                          ...
1                              Toy Story (1995)  01-Jan-1995  ...    0       0
2                              GoldenEye (1995)  01-Jan-1995  ...    0       0
3                             Four Rooms (1995)  01-Jan-1995  ...    0       0
4                             Get Shorty (1995)  01-Jan-1995  ...    0       0
5                                Copycat (1995)  01-Jan-1995  ...    0       0
...
[1682 rows x 23 columns]
```

class lenskit.datasets.**ML1M**(*path='data/ml-1m'*)

Bases: lenskit.datasets.MLM

MovieLens 1M data set.

Note: Some documentation examples use ML-10M100K; that is because this class shares implementation with the 10M data set.

---

property users

Return the movie data (from `users.dat`). Indexed by user ID.

```
>>> ml = ML1M()
>>> ml.users
     gender  age    zip
user
1         F    1  48067
2         M   56  70072
3         M   25  55117
4         M   45  02460
5         M   25  55455
...
[6040 rows x 3 columns]
```

property movies

Return the movie data (from `movies.dat`). Indexed by movie ID.

```
>>> ml = ML10M()
>>> ml.movies
                                                 title                        ␣
↪                              genres
```

(continues on next page)

(continued from previous page)

```
item
1                                        Toy Story (1995)
↪Adventure|Animation|Children|Comedy|Fantasy
2                                          Jumanji (1995)
↪  Adventure|Children|Fantasy
3                                 Grumpier Old Men (1995)
↪               Comedy|Romance
4                                Waiting to Exhale (1995)
↪         Comedy|Drama|Romance
5                       Father of the Bride Part II (1995)
↪                        Comedy
...
[10681 rows x 2 columns]
```

property ratings

Return the rating data (from `ratings.dat`).

```
>>> ml = ML10M()
>>> ml.ratings
           user  item  rating  timestamp
0             1   122     5.0  838985046
1             1   185     5.0  838983525
2             1   231     5.0  838983392
3             1   292     5.0  838983421
4             1   316     5.0  838983392
...
[10000054 rows x 4 columns]
```

class lenskit.datasets.**ML10M**(*path='data/ml-10M100K'*)

Bases: lenskit.datasets.MLM

MovieLens 10M100K data set.

property movies

Return the movie data (from `movies.dat`). Indexed by movie ID.

```
>>> ml = ML10M()
>>> ml.movies
                                                    title
↪                       genres
item
1                                        Toy Story (1995)
↪Adventure|Animation|Children|Comedy|Fantasy
2                                          Jumanji (1995)
↪  Adventure|Children|Fantasy
3                                 Grumpier Old Men (1995)
↪               Comedy|Romance
4                                Waiting to Exhale (1995)
↪         Comedy|Drama|Romance
5                       Father of the Bride Part II (1995)
↪                        Comedy
...
[10681 rows x 2 columns]
```

property ratings

Return the rating data (from `ratings.dat`).

```
>>> ml = ML10M()
>>> ml.ratings
           user  item  rating  timestamp
0             1   122     5.0  838985046
1             1   185     5.0  838983525
2             1   231     5.0  838983392
3             1   292     5.0  838983421
4             1   316     5.0  838983392
...
[10000054 rows x 4 columns]
```

1.5 Splitting Data

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

1.5.1 Row-based splitting

The simplest preparation methods sample or partition the rows in the input frame. A 5-fold *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*, ***, *rng_spec=None*)

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*) – Ratings or other data you wish to partition.
- **partitions** (*int*) – The number of partitions to produce.
- **rng_spec** – The random number generator or seed (see *lenskit.util.rng()*).

Returns an iterator of train-test pairs

Return type iterator

lenskit.crossfold.**sample_rows**(*data*, *partitions*, *size*, *disjoint=True*, ***, *rng_spec=None*)

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).

We can loop over a sequence of train-test pairs:

```
>>> from lenskit import datasets
>>> ratings = datasets.MovieLens('data/ml-latest-small').ratings
>>> for train, test in sample_rows(ratings, 5, 1000):
...     print(len(test))
1000
1000
1000
1000
1000
```

Sometimes for testing, it is useful to just get a single pair:

```
>>> train, test = sample_rows(ratings, None, 1000)
>>> len(test)
1000
>>> len(test) + len(train) - len(ratings)
0
```

Parameters

- **data** (*pandas.DataFrame*) – Data frame containing ratings or other data to partition.
- **partitions** (*int or None*) – The number of partitions to produce. If `None`, produce a _single_ train-test pair instead of an iterator or list.
- **size** (*int*) – The size of each sample.
- **disjoint** (*bool*) – If `True`, force samples to be disjoint.
- **rng_spec** – The random number generator or seed (see *lenskit.util.rng()*).

Returns An iterator of train-test pairs.

Return type iterator

1.5.2 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, ***, *rng_spec=None*)

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*) – a data frame containing ratings or other data you wish to partition.
- **partitions** (*int*) – the number of partitions to produce
- **method** (PartitionMethod) – The method for selecting test rows for each user.
- **rng_spec** – The random number generator or seed (see *lenskit.util.rng()*).

Returns iterator: an iterator of train-test pairs

lenskit.crossfold.**sample_users**(*data*, *partitions: int*, *size: int*, *method: lenskit.crossfold.PartitionMethod*, *disjoint=True*, ***, *rng_spec=None*)

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*) – Data frame containing ratings or other data you wish to partition.
- **partitions** (*int*) – The number of partitions.
- **size** (*int*) – The sample size.
- **method** (PartitionMethod) – The method for obtaining user test ratings.
- **rng_spec** – The random number generator or seed (see *lenskit.util.rng()*).

Returns An iterator of train-test pairs (as *TTPair* objects).

Return type iterator

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*, *rng_spec=None*)

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

Parameters

- **n** (*int*) – the number of test items to select
- **rng** – the random number generator or seed

lenskit.crossfold.**SampleFrac**(*frac*, *rng_spec=None*)

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

Parameters **frac** (*float*) – the fraction 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** (*int*) – The number of test items to select.

lenskit.crossfold.**LastFrac**(*frac*, *col='timestamp'*)
Select a fraction of test rows per user/item.

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

1.5.3 Utility Classes

class lenskit.crossfold.**PartitionMethod**
Bases: abc.ABC

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.

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

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

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

Return type pandas.DataFrame

__weakref__
list of weak references to the object (if defined)

class lenskit.crossfold.**TTPair**(*train*, *test*)
Bases: tuple

Train-test pair (named tuple).

property **test**
Test data for this pair.

property **train**
Train data for this pair.

1.6 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.

The batch functions can parallelize over users with the optional `n_jobs` parameter, or the `LK_NUM_PROCS` environment variable.

Note: Scripts calling the batch recommendation or prediction facilites must be *protected*; that is, they should not directly perform their work when run, but should define functions and call a `main` function when run as a script, with a block like this at the end of the file:

```
def main():
    # do the actual work

if __name__ == '__main__':
    main()
```

If you are using the batch functions from a Jupyter notbook, you should be fine - the Jupyter programs are appropriately protected.

1.6.1 Recommendation

lenskit.batch.**recommend**(*algo*, *users*, *n*, *candidates=None*, ***, *n_jobs=None*, ***kwargs*)

Batch-recommend for multiple users. The provided algorithm should be a `algorithms.Recommender`.

Parameters

- **algo** – the algorithm
- **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. Pass `None` to use the recommender's built-in candidate selector (usually recommended).
- **n_jobs** (*int*) – The number of processes to use for parallel recommendations. Passed to *lenskit.util.parallel.invoker()*.

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

1.6.2 Rating Prediction

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

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.

To use this function, provide a pre-fit algorithm:

```
>>> from lenskit.algorithms.basic import Bias
>>> from lenskit.metrics.predict import rmse
>>> from lenskit import datasets
>>> ratings = datasets.MovieLens('data/ml-latest-small').ratings
>>> bias = Bias()
>>> bias.fit(ratings[:-1000])
<lenskit.algorithms.basic.Bias object at ...>
>>> preds = predict(bias, ratings[-1000:])
>>> preds.head()
       user  item  rating   timestamp  prediction
99004   664  8361     3.0  1393891425    3.288286
99005   664  8528     3.5  1393891047    3.559119
99006   664  8529     4.0  1393891173    3.573008
99007   664  8636     4.0  1393891175    3.846268
99008   664  8641     4.5  1393890852    3.710635
>>> rmse(preds['prediction'], preds['rating'])
0.8326992222...
```

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.
- **n_jobs** (*int*) – The number of processes to use for parallel batch prediction. Passed to *lenskit.util.parallel.invoker()*.

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

1.6.3 Isolated Training

This function isn't a batch function per se, as it doesn't perform multiple operations, but it is primarily useful with batch operations. The *`train_isolated()`* function trains an algorithm in a subprocess, so all temporary resources are released by virtue of the training process exiting. It returns a shared memory serialization of the trained model, which can be passed directly to *`recommend()`* or *`predict()`* in lieu of an algorithm object, to reduce the total memory consumption.

Example usage:

```
algo = BiasedMF(50)
algo = Recommender.adapt(algo)
algo = batch.train_isolated(algo, train_ratings)
preds = batch.predict(algo, test_ratings)
```

`lenskit.batch.`**`train_isolated`**(*algo*, *ratings*, *, *file=None*, ***kwargs*)

Train an algorithm in a subprocess to isolate the training process. This function spawns a subprocess (in the same way that LensKit's multiprocessing support does), calls `lenskit.algorithms.Algorithm.fit()` on it, and serializes the result for shared-memory use.

Training the algorithm in a single-purpose subprocess makes sure that any training resources, such as TensorFlow sessions, are cleaned up by virtue of the process terminating when model training is completed. It can also reduce memory use, because the original trained model and the shared memory version are not in memory at the same time. While the batch functions use shared memory to reduce memory overhead for parallel processing, naive use of these functions will still have 2 copies of the model in memory, the shared one and the original, because the sharing process does not tear down the original model. Training in a subprocess solves this problem elegantly.

Parameters

- **`algo`** (`lenskit.algorithms.Algorithm`) – The algorithm to train.
- **`ratings`** (*`pandas.DataFrame`*) – The rating data.
- **`file`** (*`str or pathlib.Path or None`*) – The file in which to save the trained model. If `None`, uses a default file path or shared memory.
- **`kwargs`** (*`dict`*) – Additional named parameters to `lenskit.algorithms.Algorithm.fit()`.

Returns The saved model object. This is the owner, so it needs to be closed when finished to free resources.

Return type lenskit.sharing.PersistedObject

1.6.4 Scripting Evaluation

The *MultiEval* class is useful to build scripts that evaluate multiple algorithms or algorithm variants, simultaneously, across multiple data sets. It can extract parameters from algorithms and include them in the output, useful for hyperparameter search.

For example:

```
from lenskit.batch import MultiEval
from lenskit.crossfold import partition_users, SampleN
from lenskit.algorithms import basic, als
from lenskit.datasets import MovieLens
from lenskit import topn
import pandas as pd

ml = MovieLens('ml-latest-small')

eval = MultiEval('my-eval', recommend=20)
eval.add_datasets(partition_users(ml.ratings, 5, SampleN(5)), name='ML-Small')
eval.add_algorithms(basic.Popular(), name='Pop')
eval.add_algorithms([als.BiasedMF(f) for f in [20, 30, 40, 50]],
                    attrs=['features'], name='ALS')
eval.run()
```

The `my-eval/runs.csv` file will then contain the results of running these algorithms on this data set. A more complete example is available in the MultiEval notebook.

class lenskit.batch.**MultiEval**(*path*, ***, *predict=True*, *recommend=100*, *candidates=None*, *save_models=False*, *eval_n_jobs=None*, *combine=True*, ***kwargs*)

Bases: `object`

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. Any false-y value (`None`, `False`, `0`) will disable top-n. The literal value `True` will generate recommendation lists of unlimited size.
- **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. Pass `None` to use the default candidate set configured for each algorithm (recommended).
- **save_models** (*bool or str*) – save individual estimated models to disk. If `True`, models are pickled to `.pkl` files; if `'gzip'`, they are pickled to gzip-compressed `.pkl.gz` files.
- **eval_n_jobs** (*int or None*) – Value to pass to the `n_jobs` parameter in *lenskit.batch.predict()* and *lenskit.batch.recommend()*.
- **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 predictions).

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

Add one or more algorithms to the run.

Parameters

- **algos** (*algorithm or list*) – the algorithm(s) to add.
- **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.

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_count()
Get the number of runs in this evaluation.

run(*runs=None*, *, *progress=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.
- **progress** – A `tqdm.tqdm()`-compatible progress function.

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

1.7 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.

1.7.1 Prediction Accuracy Metrics

The *lenskit.metrics.predict* module contains prediction accuracy metrics. These are intended to be used as a part of a Pandas split-apply-combine operation on a data frame that contains both predictions and ratings; for convenience, the *lenskit.batch.predict()* function will include ratings in the prediction frame when its input user-item pairs contains ratings. So you can perform the following to compute per-user RMSE over some predictions:

```
from lenskit.datasets import MovieLens
from lenskit.algorithms.basic import Bias
from lenskit.batch import predict
from lenskit.metrics.predict import rmse
ratings = MovieLens('ml-small').ratings.sample(frac=10
test = ratings.iloc[:1000]
train = ratings.iloc[1000:]
algo = Bias()
algo.fit(train)
preds = predict(algo, pairs)
user_rmse = preds.groupby('user').apply(lambda df: rmse(df.prediction, df.rating))
user_rmse.mean()
```

Metric Functions

Prediction metric functions take two series, *predictions* and *truth*.

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)
```

1.7.2 Top-*N* Evaluation

LensKit's support for top-*N* evaluation is in two parts, because there are some subtle complexities that make it more dfficult to get the right data in the right place for computing metrics correctly.

Top-*N* Analysis

The *`lenskit.topn`* module contains the utilities for carrying out top-*N* analysis, in conjucntion with *`lenskit.batch.recommend()`* and its wrapper in *`lenskit.batch.MultiEval`*.

The entry point to this is *`RecListAnalysis`*. This class encapsulates an analysis with one or more metrics, and can apply it to data frames of recommendations. An analysis requires two data frames: the recommendation frame contains the recommendations themselves, and the truth frame contains the ground truth data for the users. The analysis is flexible with regards to the columns that identify individual recommendation lists; usually these will consist of a user ID, data set identifier, and algorithm identifier(s), but the analysis is configurable and its defaults make minimal assumptions. The recommendation frame does need an `item` column with the recommended item IDs, and it should be in order within a single recommendation list.

The truth frame should contain (a subset of) the columns identifying recommendation lists, along with `item` and, if available, `rating` (if no rating is provided, the metrics that need a rating value will assume a rating of 1 for every item present). It can contain other items that custom metrics may find useful as well.

For example, a recommendation frame may contain:

- DataSet
- Partition
- Algorithm
- user
- item
- rank
- score

And the truth frame:

- DataSet
- user
- item
- rating

The analysis will use this truth as the relevant item data for measuring the accuracy of the roecommendation lists. Recommendations will be matched to test ratings by data set, user, and item, using *RecListAnalysis* defaults.

class lenskit.topn.**RecListAnalysis**(*group_cols=None*, *n_jobs=None*)
Bases: object

Compute one or more top-N metrics over recommendation lists.

This method groups the recommendations by the specified columns, and computes the metric over each group. The default set of grouping columns is all columns *except* the following:

- item
- rank
- score
- rating

The truth frame, truth, is expected to match over (a subset of) the grouping columns, and contain at least an item column. If it also contains a rating column, that is used as the users' rating for metrics that require it; otherwise, a rating value of 1 is assumed.

Warning: Currently, RecListAnalysis will silently drop users who received no recommendations. We are working on an ergonomic API for fixing this problem.

Parameters **group_cols** (*list*) – The columns to group by, or None to use the default.

add_metric(*metric*, *, *name=None*, ***kwargs*)
Add a metric to the analysis.

A metric is a function of two arguments: the a single group of the recommendation frame, and the corresponding truth frame. The truth frame will be indexed by item ID. The recommendation frame will be in the order in the data. Many metrics are defined in *lenskit.metrics.topn*; they are re-exported from *lenskit.topn* for convenience.

Parameters

- **metric** – The metric to compute.
- **name** – The name to assign the metric. If not provided, the function name is used.
- ****kwargs** – Additional arguments to pass to the metric.

compute(*recs*, *truth*, *, *include_missing=False*)
Run the analysis. Neither data frame should be meaningfully indexed.

Parameters

- **recs** (*pandas.DataFrame*) – A data frame of recommendations.
- **truth** (*pandas.DataFrame*) – A data frame of ground truth (test) data.

- **include_missing** (*bool*) – `True` to include users from truth missing from recs. Matches are done via group columns that appear in both `recs` and `truth`.

Returns The results of the analysis.

Return type pandas.DataFrame

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*, *truth*)
Compute recommendation precision.

`lenskit.metrics.topn.`**`recall`**(*recs*, *truth*)
Compute recommendation recall.

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*, *truth*)
Compute the reciprocal rank of the first relevant item in a list of recommendations.

If no elements are relevant, the reciprocal rank is 0.

Utility Metrics

The NDCG function estimates a utility score for a ranked list of recommendations.

`lenskit.metrics.topn.`**`ndcg`**(*recs*, *truth*, *discount=<ufunc 'log2'>*)
Compute the normalized discounted cumulative gain.

Discounted cumultative gain is computed as:

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

This is then normalized as follows:

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

Parameters

- **recs** – The recommendation list.
- **truth** – The user's test data.
- **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.

We also expose the internal DCG computation directly.

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.

This is not a true top-N metric, but is a utility function for other metrics.

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

1.7.3 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()
```

1.8 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()`.

1.8.1 Serialization

Like SciKit models, all LensKit algorithms are pickleable, and this is how we recommend saving models to disk for later use. This can be done with `pickle`, but we recommend using `binpickle` for more automatically-optimized storage. For example, to save a fully-configured ALS module with fairly aggressive ZSTD compression:

```
algo = Recommender.adapt(ImplicitMF(50))
algo.fit(ratings)
binpickle.dump(algo, binpickle.codecs.Blosc('zstd', 9))
```

1.8.2 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.**Algorithm**
Bases: `object`

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

Canonical lenskit.Algorithm

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

class algorithms.**Algorithm**
This is an alias of *`lenskit.Algorithm`*.

1.8.3 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)
```

If the algorithm already implements *Recommender*, it is returned, so it is safe to always call *Recommender.adapt()* before fitting an algorithm you will need for top-*N* recommendations to mak sure it is suitable.

class lenskit.**Recommender**

Bases: *lenskit.algorithms.Algorithm*

Recommends lists of items for users.

abstract 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; if `None`, a default set will be used. For many algorithms, this is their *CandidateSelector*.
- **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

classmethod adapt(*algo*)

Ensure that an algorithm is a *Recommender*. If it is not a recommender, it is wrapped in a `lenskit.basic.TopN` with a default candidate selector.

Note: Since 0.6.0, since algorithms are fit directly, you should call this method **before** calling *Algorithm.fit()*, unless you will always be passing explicit candidate sets to *recommend()*.

Parameters **algo** (`Predictor`) – the underlying rating predictor.

class algorithms.**Recommender**

This is an alias of *lenskit.Recommender*.

1.8.4 Candidate Selection

Some recommenders use a *candidate selector* to identify possible items to recommend. These are also treated as algorithms, mainly so that they can memorize users' prior ratings to exclude them from recommendation.

class lenskit.**CandidateSelector**
Bases: *lenskit.algorithms.Algorithm*

Select candidates for recommendation for a user, possibly with some additional ratings.

UnratedItemCandidateSelector is the default and most common implementation of this interface.

abstract candidates(*user*, *ratings=None*)
Select candidates for the user.

Parameters

- **user** – The user key or ID.
- **ratings** (*pandas.Series or array-like*) – Ratings or items to use instead of whatever ratings were memorized for this user. If a `pandas.Series`, the series index is used; if it is another array-like it is assumed to be an array of items.

static rated_items(*ratings*)
Utility function for converting a series or array into an array of item IDs. Useful in implementations of *candidates()*.

class algorithms.**CandidateSelector**
This is an alias of *lenskit.CandidateSelector*.

1.8.5 Rating Prediction

The *Predictor* class impelemnts 'rating prediction', as well as any other personalized item scoring that may not be predictions of actual ratings. Most algorithms actually implement this interface.

class lenskit.**Predictor**
Bases: *lenskit.algorithms.Algorithm*

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

abstract 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

class algorithms.**Predictor**
This is an alias of *lenskit.Predictor*.

1.9 Algorithm Summary

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.

All algorithms implement the standard interfaces.

1.9.1 Basic Algorithms

basic.Bias([items, users, damping])	A user-item bias rating prediction algorithm.
basic.Popular([selector])	Recommend the most popular items.
basic.TopN(predictor[, selector])	Basic recommender that implements top-N recommendation using a predictor.
basic.Fallback(algorithms, *others)	The Fallback algorithm predicts with its first component, uses the second to fill in missing values, and so forth.
basic.UnratedItemCandidateSelector()	*CandidateSelector* that selects items a user has not rated as candidates.
basic.Memorized(scores)	The memorized algorithm memorizes socres provided at construction time.

1.9.2 k-NN Algorithms

user_knn.UserUser(nnbrs[, min_nbrs, ...])	User-user nearest-neighbor collaborative filtering with ratings.
item_knn.ItemItem(nnbrs[, min_nbrs, ...])	Item-item nearest-neighbor collaborative filtering with ratings.

1.9.3 Matrix Factorization

als.BiasedMF(features, *[, iterations, reg, ...])	Biased matrix factorization trained with alternating least squares [ZWSP2008].
als.ImplicitMF(features, *[, iterations, ...])	Implicit matrix factorization trained with alternating least squares [HKV2008].
funksvd.FunkSVD(features[, iterations, ...])	Algorithm class implementing FunkSVD matrix factorization.

1.9.4 TensorFlow

tf.BiasedMF([features, bias, damping, ...])	Biased matrix factorization model for explicit feedback, optimized with TensorFlow.
tf.IntegratedBiasMF([features, epochs, ...])	Biased matrix factorization model for explicit feedback, optimizing both bias and embeddings with TensorFlow.
tf.BPR([features, epochs, batch_size, reg, ...])	Bayesian Personalized Ranking with matrix factorization, optimized with TensorFlow.

1.9.5 External Library Wrappers

implicit.BPR(*args, **kwargs)	LensKit interface to `implicit.bpr`.
implicit.ALS(*args, **kwargs)	LensKit interface to `implicit.als`.
hpf.HPF(features, **kwargs)	Hierarchical Poisson factorization, provided by hpfrec.

1.9.6 References

1.10 Basic and Utility Algorithms

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

1.10.1 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 μ is the global mean rating, b_i is item bias, and b_u is the user bias. With the provided damping values β_u and β_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_: double
The global mean rating.

item_offsets_: pandas.Series
The item offsets (b_i values)

user_offsets_: pandas.Series
The item offsets (b_u values)

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

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

Returns the fit bias object.

Return type *Bias*

transform(*ratings*, *, *indexes=False*)
Transform ratings by removing the bias term.

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings to transform. Must contain at least `user`, `item`, and `rating` columns.
- **indexes** (*bool*) – if `True`, the resulting frame will include `uidx` and `iidx` columns containing the 0-based user and item indexes for each rating.

Returns A data frame with `rating` transformed by subtracting user-item bias prediction.

Return type pandas.DataFrame

inverse_transform(*ratings*)
Transform ratings by removing the bias term.

fit_transform(*ratings*, ***kwargs*)
Fit with ratings and return the training data transformed.

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

property user_index
Get the user index from this (fit) bias.

property item_index
Get the item index from this (fit) bias.

1.10.2 Most Popular Item Recommendation

The *Popular* algorithm implements most-popular-item recommendation.

class lenskit.algorithms.basic.**Popular**(*selector=None*)
Bases: *lenskit.algorithms.Recommender*

Recommend the most popular items.

Parameters **selector** (CandidateSelector) – The candidate selector to use. If None, uses a new *UnratedItemCandidateSelector*.

item_pop_: pandas.Series
Item rating counts (popularity)

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

Returns The algorithm object.

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; if None, a default set will be used. For many algorithms, this is their CandidateSelector.
- **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

1.10.3 Random Item Recommendation

The *Random* algorithm implements random-item recommendation.

class lenskit.algorithms.basic.**Random**(*selector=None*, *rng_spec=None*)
Bases: *lenskit.algorithms.Recommender*

A random-item recommender.

selector: CandidateSelector
Selects candidate items for recommendation. Default is *UnratedItemCandidateSelector*.

rng_spec
Seed or random state for generating recommendations. Pass 'user' to deterministically derive per-user RNGS from the user IDs for reproducibility.

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

Returns The algorithm object.

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; if `None`, a default set will be used. For many algorithms, this is their `CandidateSelector`.
- **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

1.10.4 Top-N Recommender

The *TopN* class implements a standard top-*N* recommender that wraps a *Predictor* and *CandidateSelector* and returns the top *N* candidate items by predicted rating. It is the type of recommender returned by *Recommender.adapt()* if the provided algorithm is not a recommender.

class lenskit.algorithms.basic.**TopN** (*predictor*, *selector=None*)
Bases: *lenskit.algorithms.Recommender*, *lenskit.algorithms.Predictor*

Basic recommender that implements top-N recommendation using a predictor.

Note: This class does not do anything of its own in *fit()*. If its predictor and candidate selector are both fit, the top-N recommender does not need to be fit.

Parameters

- **predictor** (`Predictor`) – The underlying predictor.
- **selector** (`CandidateSelector`) – The candidate selector. If `None`, uses *UnratedItemCandidateSelector*.

fit (*ratings*, ***kwargs*)
Fit the recommender.

Parameters

- **ratings** (*pandas.DataFrame*) – The rating or interaction data. Passed changed to the predictor and candidate selector.

- **kwargs** (*args,*) – Additional arguments for the predictor to use in its training process.

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; if `None`, a default set will be used. For many algorithms, this is their `CandidateSelector`.
- **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

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

1.10.5 Unrated Item Candidate Selector

UnratedItemCandidateSelector is a candidate selector that remembers items users have rated, and returns a candidate set consisting of all unrated items. It is the default candidate selector for *TopN*.

class lenskit.algorithms.basic.**UnratedItemCandidateSelector**
Bases: *lenskit.algorithms.CandidateSelector*

`CandidateSelector` that selects items a user has not rated as candidates. When this selector is fit, it memorizes the rated items.

items_: pandas.Index
All known items.

users_: pandas.Index
All known users.

user_items_: CSR
Items rated by each known user, as positions in the `items` index.

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

Returns The algorithm object.

candidates(*user*, *ratings=None*)
Select candidates for the user.

Parameters

- **user** – The user key or ID.
- **ratings** (*pandas.Series or array-like*) – Ratings or items to use instead of whatever ratings were memorized for this user. If a `pandas.Series`, the series index is used; if it is another array-like it is assumed to be an array of items.

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

1.10.7 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.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

1.11 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.

1.11.1 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.

Parameters

- **nnbrs** (*int*) – the maximum number of neighbors for scoring each item (`None` for unlimited)
- **min_nbrs** (*int*) – the minimum number of neighbors for scoring each item
- **min_sim** (*double*) – minimum similarity threshold for considering a neighbor
- **save_nbrs** (*double*) – the number of neighbors to save per item in the trained model (`None` for unlimited)
- **center** (*bool*) – whether to normalize (mean-center) rating vectors. Turn this off when working with unary data and other data types that don't respond well to centering.
- **aggregate** – the type of aggregation to do. Can be `weighted-average` or `sum`.

item_index_: pandas.Index
the index of item IDs.

item_means_: numpy.ndarray
the mean rating for each known item.

item_counts_: numpy.ndarray
the number of saved neighbors for each item.

sim_matrix_: matrix.CSR
the similarity matrix.

user_index_: pandas.Index
the index of known user IDs for the rating matrix.

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

fit(*ratings*, ***kwargs*)
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.

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

1.11.2 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.

Parameters

- **nnbrs** (*int*) – the maximum number of neighbors for scoring each item (`None` for unlimited)
- **min_nbrs** (*int*) – the minimum number of neighbors for scoring each item
- **min_sim** (*double*) – minimum similarity threshold for considering a neighbor
- **center** (*bool*) – whether to normalize (mean-center) rating vectors. Turn this off when working with unary data and other data types that don't respond well to centering.
- **aggregate** – the type of aggregation to do. Can be `weighted-average` or `sum`.

user_index_: pandas.Index
User index.

item_index_: pandas.Index
Item index.

user_means_: numpy.ndarray
User mean ratings.

rating_matrix_: matrix.CSR
Normalized user-item rating matrix.

transpose_matrix_: matrix.CSR
Transposed un-normalized rating matrix.

fit(*ratings*, ***kwargs*)
"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

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

1.12 Classic Matrix Factorization

LKPY provides classical matrix factorization implementations.

1.12.1 Common Support

The `mf_common` module contains common support code for matrix factorization algorithms. These classes, *`MFPredictor`* and *`BiasMFPredictor`*, define the parameters that are estimated during the *`Algorithm.fit()`* process on common matrix factorization algorithms.

class `lenskit.algorithms.mf_common.`**MFPredictor**

Bases: *`lenskit.algorithms.Predictor`*

Common predictor for matrix factorization.

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

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

user_features_: numpy.ndarray
The $m \times k$ user-feature matrix.

item_features_: numpy.ndarray
The $n \times k$ item-feature matrix.

property n_features
The number of features.

property n_users
The number of users.

property n_items
The number of items.

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

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

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**
Bases: *lenskit.algorithms.mf_common.MFPredictor*

Common model for biased matrix factorization.

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

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

global_bias_: double
The global bias term.

user_bias_: numpy.ndarray
The user bias terms.

item_bias_: numpy.ndarray
The item bias terms.

user_features_: numpy.ndarray
The $m \times k$ user-feature matrix.

item_features_: numpy.ndarray
The $n \times k$ item-feature matrix.

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

1.12.2 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.

class lenskit.algorithms.als.**BiasedMF**(*features*, ***, *iterations=20*, *reg=0.1*, *damping=5*, *bias=True*, *method='cd'*, *rng_spec=None*, *progress=None*)

Bases: *lenskit.algorithms.mf_common.BiasMFPredictor*

Biased matrix factorization trained with alternating least squares [ZWSP2008]. This is a prediction-oriented algorithm suitable for explicit feedback data.

It provides two solvers for the optimization step (the *method* parameter):

'cd' (the default) Coordinate descent [TPT2011], adapted for a separately-trained bias model and to use weighted regularization as in the original ALS paper [ZWSP2008].

'lu' A direct implementation of the original ALS concept [ZWSP2008] using LU-decomposition to solve for the optimized matrices.

See the base class *BiasMFPredictor* for documentation on the estimated parameters you can extract from a trained model.

Parameters

- **features** (*int*) – the number of features to train
- **iterations** (*int*) – the number of iterations to train
- **reg** (*float*) – the regularization factor; can also be a tuple `(ureg, ireg)` to specify separate user and item regularization terms.
- **damping** (*float*) – damping factor for the underlying mean
- **bias** (bool or `Bias`) – the bias model. If `True`, fits a `Bias` with damping `damping`.
- **method** (*str*) – the solver to use (see above).
- **rng_spec** – Random number generator or state (see *lenskit.util.random.rng()*).
- **progress** – a `tqdm.tqdm()`-compatible progress bar function

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

Run ALS to train a model.

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

Returns The algorithm (for chaining).

fit_iters(*ratings*, ***kwargs*)

Run ALS to train a model, returning each iteration as a generator.

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

Returns The algorithm (for chaining).

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

class lenskit.algorithms.als.**ImplicitMF**(*features*, ***, *iterations=20*, *reg=0.1*, *weight=40*, *method='cg'*, *rng_spec=None*, *progress=None*)

Bases: *lenskit.algorithms.mf_common.MFPredictor*

Implicit matrix factorization trained with alternating least squares [HKV2008]. This algorithm outputs 'predictions', but they are not on a meaningful scale. If its input data contains `rating` values, these will be used as the 'confidence' values; otherwise, confidence will be 1 for every rated item.

`'cd'` (the default) Conjugate gradient method [TPT2011].

`'lu'` A direct implementation of the original implicit-feedback ALS concept [HKV2008] using LU-decomposition to solve for the optimized matrices.

See the base class *MFPredictor* for documentation on the estimated parameters you can extract from a trained model.

Parameters

- **features** (*int*) – the number of features to train
- **iterations** (*int*) – the number of iterations to train
- **reg** (*double*) – the regularization factor
- **weight** (*double*) – the scaling weight for positive samples (α in [HKV2008]).
- **rng_spec** – Random number generator or state (see *lenskit.util.random.rng()*).
- **progress** – a `tqdm.tqdm()`-compatible progress bar function

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

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

1.12.3 SciKit SVD

This code implements a traditional SVD using scikit-learn. It requires `scikit-learn` to be installed in order to function.

class lenskit.algorithms.svd.**BiasedSVD**(*features*, *, *damping=5*, *bias=True*, *algorithm='randomized'*)

Bases: *lenskit.algorithms.Predictor*

Biased matrix factorization for implicit feedback using SciKit-Learn's SVD solver (`sklearn.decomposition.TruncatedSVD`). It operates by first computing the bias, then computing the SVD of the bias residuals.

You'll generally want one of the iterative SVD implementations such as `lennskit.algorithms.als.BiasedMF`; this is here primarily as an example and for cases where you want to evaluate a pure SVD implementation.

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

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

1.12.4 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. We generally don't recommend using it in new applications or experiments; the ALS-based algorithms are less sensitive to hyperparameters, and the TensorFlow algorithms provide more optimized gradient descent training of the same prediction model.

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

Bases: *lenskit.algorithms.mf_common.BiasMFPredictor*

Algorithm class implementing FunkSVD matrix factorization. FunkSVD is a regularized biased matrix factorization technique trained with featurewise stochastic gradient descent.

See the base class *BiasMFPredictor* for documentation on the estimated parameters you can extract from a trained model.

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.
- **random_state** – The random state for shuffling the data prior to training.

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

1.13 TensorFlow Algorithms

LKPY provides several algorithm implementations, particularly matrix factorization, using TensorFlow. These algorithms serve two purposes:

- Provide classic algorithms ready to use for recommendation or as baselines for new techniques.
- Demonstrate how to connect TensorFlow to LensKit for use in your own experiments.

1.13.1 Biased MF

These models implement the standard biased matrix factorization model, like *lenskit.algorithms.als.BiasedMF*, but learn the model parameters using TensorFlow's gradient descent instead of the alternating least squares algorithm.

Bias-Based

class lenskit.algorithms.tf.**BiasedMF**(*features=50*, ***, *bias=True*, *damping=5*, *epochs=5*, *batch_size=10000*, *reg=0.02*, *rng_spec=None*)

Bases: *lenskit.algorithms.mf_common.BiasMFPredictor*

Biased matrix factorization model for explicit feedback, optimized with TensorFlow.

This is a basic TensorFlow implementation of the biased matrix factorization model for rating prediction:

$$s(i|u) = b + b_u + b_i + \vec{p}_u \cdot \vec{q}_i$$

User and item embedding matrices are regularized with L_2 regularization, governed by a regularization term λ. Regularizations for the user and item embeddings are then computed as follows:

$$\lambda_u = \lambda/|U|$$
$$\lambda_i = \lambda/|I|$$

This rescaling allows the regularization term to be independent of the number of users and items.

Because the model is very simple, this algorithm works best with large batch sizes.

This implementation uses *lenskit.algorithms.basic.Bias* for computing the biases, and uses TensorFlow to fit a matrix factorization on the residuals. It then extracts the resulting matrices, and relies on `BiasedMFPredictor` to implement the prediction logic, like *lenskit.algorithms.als.BiasedMF*. Its code is suitable as an example of how to build a Keras/TensorFlow algorithm implementation for LensKit where TF is only used in the train stage.

A variety of resources informed the design, most notably this one.

Parameters

- **features** (*int*) – The number of latent features to learn.
- **bias** – The bias model to use.
- **damping** – The bias damping, if `bias` is `True`.
- **epochs** (*int*) – The number of epochs to train.
- **batch_size** (*int*) – The Keras batch size.
- **reg** (*double*) – The regularization term λ used to derive embedding vector regularizations.
- **rng_spec** – The random number generator initialization.

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

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

Fully Integrated

class lenskit.algorithms.tf.**IntegratedBiasMF**(*features=50*, ***, *epochs=5*, *batch_size=10000*, *reg=0.02*, *bias_reg=0.2*, *rng_spec=None*)

Bases: *lenskit.algorithms.Predictor*

Biased matrix factorization model for explicit feedback, optimizing both bias and embeddings with TensorFlow.

This is a basic TensorFlow implementation of the biased matrix factorization model for rating prediction:

$$s(i|u) = b + b_u + b_i + \vec{p}_u \cdot \vec{q}_i$$

User and item embedding matrices are regularized with L_2 regularization, governed by a regularization term λ. Regularizations for the user and item embeddings are then computed as follows:

$$\lambda_u = \lambda/|U|$$
$$\lambda_i = \lambda/|I|$$

This rescaling allows the regularization term to be independent of the number of users and items. The same rescaling applies to the bias regularization.

Because the model is very simple, this algorithm works best with large batch sizes.

This implementation uses TensorFlow to fit the entire model, including user/item biases and residuals, and uses TensorFlow to do the final predictions as well. Its code is suitable as an example of how to build a Keras/TensorFlow algorithm implementation for LensKit where TF used for the entire process.

A variety of resources informed the design, most notably this one and **`Chin-chi Hsu's example code`_**.

Parameters

- **features** (*int*) – The number of latent features to learn.
- **epochs** (*int*) – The number of epochs to train.
- **batch_size** (*int*) – The Keras batch size.
- **reg** (*double*) – The regularization term for the embedding vectors.
- **bias_reg** (*double*) – The regularization term for the bias vectors.
- **rng_spec** – The random number generator initialization.

model
The Keras model.

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.

- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

1.13.2 Bayesian Personalized Rating

class lenskit.algorithms.tf.**BPR**(*features=50*, ***, *epochs=5*, *batch_size=10000*, *reg=0.02*, *neg_count=1*, *rng_spec=None*)
Bases: *lenskit.algorithms.Predictor*

Bayesian Personalized Ranking with matrix factorization, optimized with TensorFlow.

This is a basic TensorFlow implementation of the BPR algorithm _[BPR].

User and item embedding matrices are regularized with L_2 regularization, governed by a regularization term λ. Regularizations for the user and item embeddings are then computed as follows:

$$\lambda_u = \lambda/|U|$$
$$\lambda_i = \lambda/|I|$$

This rescaling allows the regularization term to be independent of the number of users and items.

Because the model is relatively simple, optimization works best with large batch sizes.

Parameters

- **features** (*int*) – The number of latent features to learn.
- **epochs** (*int*) – The number of epochs to train.
- **batch_size** (*int*) – The Keras batch size. This is the number of **positive** examples to sample in each batch. If `neg_count` is greater than 1, the batch size will be similarly multipled.
- **reg** (*double*) – The regularization term for the embedding vectors.
- **neg_count** (*int*) – The number of negative examples to sample for each positive one.
- **rng_spec** – The random number generator initialization.

model
The Keras model.

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

1.14 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*)
Bases: *lenskit.algorithms.mf_common.MFPredictor*

Hierarchical Poisson factorization, provided by hpfrec.

Parameters

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

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

Parameters

- **ratings** (*pandas.DataFrame*) – The ratings data.
- **kwargs** – Additional training data the algorithm may require. Algorithms should avoid using the same keyword arguments for different purposes, so that they can be more easily hybridized.

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

1.15 Implicit

This module provides a LensKit bridge to Ben Frederickson's implicit library implementing some implicit-feedback recommender algorithms, with an emphasis on matrix factorization.

class lenskit.algorithms.implicit.**ALS**(**args*, ***kwargs*)

Bases: lenskit.algorithms.implicit.BaseRec

LensKit interface to implicit.als.

class lenskit.algorithms.implicit.**BPR**(**args*, ***kwargs*)

Bases: lenskit.algorithms.implicit.BaseRec

LensKit interface to implicit.bpr.

1.16 Performance Tips

LensKit strives to provide pretty good performance (in terms of computation speed), but sometimes it needs a little nudging.

Note: If you are implementing an algorithm, see the implementation tips for information on good performance.

1.16.1 Quick Tips

- Use Conda-based Python, with `tbb` installed.
- Set the `MKL_THREADING_LAYER` environment variable to `tbb`, so both MKL and LensKit will use TBB and can coordinate their thread pools.
- Use `LK_NUM_PROCS` if you want to control LensKit's batch prediction and recommendation parallelism, and `NUMBA_NUM_THREADS` to control its model training parallelism.

We generally find the best performance using MKL with TBB throughout the stack. If both LensKit's Numba-accelerated code and MKL are using TBB, they will coordinate their thread pools to coordinate threading levels.

1.16.2 Controlling Parallelism

LensKit has two forms of parallelism. Algorithm training processes can be parallelized through a number of mechanisms:

- Our own parallel code uses Numba, which in turn uses TBB (preferred) or OpenMP. The thread count is controlled by `NUMBA_NUM_THREADS`.
- The BLAS library may parallelize underlying operations using its threading library. This is usually OpenMP; MKL also supports TBB, but unlike Numba, it defaults to OpenMP even if TBB is available.
- Underlying libraries such as TensorFlow and scikit-learn may provide their own parallelism.

The LensKit batch functions use Python `multiprocessing`, and their concurrency level is controlled by the `LK_NUM_PROCS` environment variable. The default number of processes is one-half the number of cores as reported by `multiprocessing.cpu_count()`. The batch functions also set the thread count for some libraries within the worker processess, to prevent over-subscribing the CPU. Right now, the worker will configure Numba and MKL. In the rest of this section, this will be referred to as the 'inner thread count'.

The thread count logic is controlled by *`lenskit.util.parallel.proc_count()`*, and works as follows:

- If `LK_NUM_PROCS` is an integer, the batch functions will use the specified number of processes, and with 1 inner thread.
- If `LK_NUM_PROCS` is a comma-separated pair of integers (e.g. `8,4`), the batch functions will use the first number for the process count and the second number as the inner thread count.
- If `LK_NUM_PROCS` is not set, the batch functions use half the number of cores as the process count and 2 as the inner thread count.

1.16.3 Other Notes

- Batch parallelism **disables** TensorFlow GPUs in the worker threads. This is fine, because GPUs are most useful for model training; multiple worker processes competing for the GPU causes problems.

1.17 Errors and Diagnostics

1.17.1 Logging

LensKit algorithms and evaluation routines report diagnostic data using the standard Python `logging` framework. Loggers are named after the corresponding Python module, and all live under the `lenskit` namespace.

Algorithms usually report erroneous or anomalous conditions using Python exceptions and warnings. **Evaluation code**, such as that in *`lenskit.batch`*, typically reports such conditions using the logger, as the common use case is to be running them in a script.

1.17.2 Warnings

In addition to Python standard warning types such as `warnings.DeprecationWarning`, LensKit uses the following warning classes to report anomalous problems in use of LensKit.

class lenskit.**DataWarning**

Bases: `UserWarning`

Warning raised for detectable problems with input data.

1.18 Algorithm Implementation Tips

Implementing algorithms is fun, but there are a few things that are good to keep in mind.

In general, development follows the following:

1. Correct
2. Clear
3. Fast

In that order. Further, we always want LensKit to be *usable* in an easy fashion. Code implementing algorithms, however, may be quite complex in order to achieve good performance.

1.18.1 Performance

We use Numba to optimize critical code paths and provide parallelism in a number of cases, such as ALS training. See the ALS source code for examples.

We also directly use MKL sparse matrix routines when available for some operations. Whenever this is done in the main LensKit code base, however, we also provide fallback implementations when the MKL is not available. The k-NN recommenders both demonstrate different versions of this. The `_mkl_ops` module exposes MKL operations; we implement them through C wrappers in the `mkl_ops.c` file, that are then called through FFI. This extra layer is because the raw MKL calls are quite complex to call via FFI, and are not particularly amenable to use with Numba. We re-expose simplified interfaces that are also usable with Numba.

1.18.2 Pickling and Sharing

LensKit uses Python pickling (or JobLib's modified pickling in `joblib.dump()`) quite a bit to save and reload models and to share model data between concurrent processes. This generally just works, and you don't need to implement any particular save/load logic in order to have your algorithm be savable and sharable.

There are a few exceptions, though.

If your algorithm updates state after fitting, this should *not* be pickled. An example of this would be caching predictions or recommendations to save time in subsequent calls. Only the model parameters and estimated parameters should be pickled. If you have caches or other ephemeral structures, override `__getstate__` and `__setstate__` to exclude them from the saved data and to initialize caches to empty values on unpickling.

If your model excludes secondary data structures from pickling, such as a reverse index of user-item interactions, then you should only exclude them when pickling for serialization. When pickling for model sharing (see *lenskit.sharing.in_share_context()*), you should include the derived structures so they can also be shared.

If your algorithm uses subsidiary models as a part of the training process, but does not need them for prediction or recommendation (for example, `lenskit.algorithms.als.BiasMF`'s use of *lenskit.algorithms.basic.Bias* in `fit`, during which it copies the bias model's internal state to its own fields), then consider overriding `__getstate__` to remove the underlying model or replace it with a cloned copy (with *lenskit.util.clone()*) to reduce serialized disk space (and deserialized memory use).

1.18.3 Random Number Generation

See *lenskit.util.random* for documentation on how to use random number generation.

In general, algorithms using randomization should have an `rng` parameter that takes a seed or RNG, and pass this to *lenskit.util.random.rng()* to get a random number generator. Algorithms that use randomness at predict or recommendation time, not just training time, should support the value `'user'` for the `rng` parameter, and if it is passed, derive a new seed for each user using *lenskit.util.random.derive_seed()* to allow reproducibility in the face of parallelism for common experimental designs. *lenskit.util.random.derivable_rng()* automates this logic.

1.18.4 Memory Map Friendliness

LensKit uses `joblib.Parallel` to parallelize internal operations (when it isn't using Numba). Joblib is pretty good about using shared memory to minimize memory overhead in parallel computations, and LensKit has some tricks to maximize this use. However, it does require a bit of attention in your algorithm implementation.

The easiest way to make this fail is to use many small NumPy or Pandas data structures. If you have a dictionary of `np.ndarray` objects, for instance, it will cause a problem. This is because each array will be memory-mapped, and each map will *reopen* the file. Having too many active open files will cause your process to run out of file descriptors on many systems. Keep your object count to a small, ideally fixed number; in `lenskit.algorithms.basic.UnratedItemSelector`, we do this by storing user and item indexes along with a `matrix.CSR` containing the items rated by each user. The old implementation had a dictionary mapping user IDs to ``ndarray``s with each user's rated items. This is a change from $|U| + 1$ arrays to 5 arrays.

1.19 Utility Functions

These utility functions are useful for data processing.

1.19.1 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*, ***, *users=None*, *items=None*)

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`*.
- **`users`** (*`pandas.Index`*) – an index of user IDs.
- **`items`** (*`pandas.Index`*) – an index of items IDs.

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

Return type *RatingMatrix*

`class` `lenskit.matrix.`**`RatingMatrix`**(*matrix*, *users*, *items*)

Bases: `tuple`

A rating matrix with associated indices.

`matrix: CSR or scipy.sparse.csr_matrix`
The rating matrix, with users on rows and items on columns.

`users: pandas.Index`
mapping from user IDs to row numbers.

`items: pandas.Index`
mapping from item IDs to column numbers.

`property items`
Alias for field number 2

property matrix
Alias for field number 0

property users
Alias for field number 1

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.

class lenskit.matrix.**CSR**(*nrows=None*, *ncols=None*, *nnz=None*, *ptrs=None*, *inds=None*, *vals=None*, *N=None*)
Bases: object

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

- It is backed by 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 with the constructor. Instead, use one of its class methods.

If you need to pass an instance off to a Numba-compiled function, use *N*:

```
_some_numba_fun(csr.N)
```

We use the indirection between this and the Numba jitclass so that the main CSR implementation can be pickled, and so that we can have class and instance methods that are not compatible with jitclass but which are useful from interpreted code.

N: _CSR
the Numba jitclass backing (has the same attributes and most methods).

nrows: int
the number of rows.

ncols: int
the number of columns.

nnz: int
the number of entries.

rowptrs: numpy.ndarray
the row pointers.

colinds: numpy.ndarray
the column indices.

values: numpy.ndarray
the values

classmethod empty(*shape*, *row_nnzs*, *, *rpdtype=<class 'numpy.int32'>*)
Create an empty CSR matrix.

Parameters

- **shape** (*tuple*) – the array shape (rows,cols)

- **row_nnzs** (*array-like*) – the number of nonzero entries for each row

classmethod from_coo(*rows*, *cols*, *vals*, *shape=None*, *rpdtype=<class 'numpy.int32'>*)
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.

classmethod 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*

to_scipy()
Convert a CSR matrix to a SciPy `scipy.sparse.csr_matrix`. Avoids copying if possible.

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

Returns A SciPy sparse matrix with the same data.

Return type scipy.sparse.csr_matrix

property N
Get the native backing array.

subset_rows(*begin*, *end*)
Subset the rows in this matrix.

rowinds() → numpy.ndarray
Get the row indices from this array. Combined with *colinds* and *values*, this can form a COO-format sparse matrix.

row(*row*)
Return a row of this matrix as a dense ndarray.

Parameters **row** (*int*) – the row index.

Returns the row, with 0s in the place of missing values.

Return type numpy.ndarray

row_extent(*row*)
Get the extent of a row in the underlying column index and value arrays.

Parameters **row** (*int*) – the row index.

Returns `(s, e)`, where the row occupies positions $[s, e)$ in the CSR data.

Return type tuple

row_cs(*row*)
Get the column indcies for the stored values of a row.

row_vs(*row*)
Get the stored values of a row.

row_nnzs()
Get a vector of the number of nonzero entries in each row.

Note: This method is not available from Numba.

Returns the number of nonzero entries in each row.

Return type numpy.ndarray

normalize_rows(*normalization*)
Normalize the rows of the matrix.

Note: The normalization *ignores* missing values instead of treating them as 0.

Note: This method is not available from Numba.

Parameters **normalization** (*str*) – The normalization to perform. Can be one of:

- `'center'` - center rows about the mean
- `'unit'` - convert rows to a unit vector

Returns The normalization values for each row.

Return type numpy.ndarray

transpose(*values=True*)
Transpose a CSR matrix.

Note: This method is not available from Numba.

Parameters **values** (*bool*) – whether to include the values in the transpose.

Returns the transpose of this matrix (or, equivalently, this matrix in CSC format).

Return type *CSR*

filter_nnzs(*filt*)
Filter the values along the full NNZ axis.

Parameters **filt** (*ndarray*) – a logical array of length *nnz* that indicates the values to keep.

Returns The filtered sparse matrix.

Return type *CSR*

class lenskit.matrix.**_CSR**(*nrows*, *ncols*, *nnz*, *ptrs*, *inds*, *vals*)
Bases: object

Internal implementation class for *CSR*. If you work with CSRs from Numba, you will use a `numba.jitclass()`-ed version of this.

Note that the `values` array is always present (unlike the Python shim), but is zero-length if no values are present. This eases Numba type-checking.

1.19.2 Math utilities

Solvers

`lenskit.math.solve.`**`dposv`**(*A*, *b*, *lower=False*)
Interface to the BLAS dposv function. A Numba-accessible verison without error checking is exposed as *`_dposv()`*.

`lenskit.math.solve.`**`solve_tri`**(*A*, *b*, *transpose=False*, *lower=True*)
Solve the system $Ax = b$, where A is triangular. This is equivalent to `scipy.linalg.solve_triangular()`, but does *not* check for non-singularity. It is a thin wrapper around the BLAS `dtrsv` function.

Parameters

- **`A`** (*`ndarray`*) – the matrix.
- **`b`** (*`ndarray`*) – the taget vector.
- **`transpose`** (*`bool`*) – whether to solve $Ax = b$ or $A^T x = b$.
- **`lower`** (*`bool`*) – whether A is lower- or upper-triangular.

Numba-accessible internals

`lenskit.math.solve.`**`_dposv`**(*A*, *b*, *lower*)

`lenskit.math.solve.`**`_dtrsv`**(*lower*, *trans*, *a*, *x*)

1.19.3 Miscellaneous

Miscellaneous utility functions.

`lenskit.util.`**`log_to_stderr`**(*level=20*)
Set up the logging infrastructure to show log output on `sys.stderr`, where it will appear in the IPython message log.

`lenskit.util.`**`log_to_notebook`**(*level=20*)
Set up the logging infrastructure to show log output in the Jupyter notebook.

`class` `lenskit.util.`**`Stopwatch`**(*start=True*)
Bases: `object`

Timer class for recording elapsed wall time in operations.

`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.`**`rng`**(*spec=None*, ***, *legacy=False*)
Get a random number generator. This is similar to `sklearn.utils.check_random_seed()`, but it usually returns a `numpy.random.Generator` instead.

Parameters

- **spec** – The spec for this RNG. Can be any of the following types:
  - `int`
  - `None`
  - `numpy.random.SeedSequence`
  - `numpy.random.mtrand.RandomState`
  - `numpy.random.Generator`
- **legacy** (*bool*) – If `True`, return `numpy.random.mtrand.RandomState` instead of a new-style `numpy.random.Generator`.

Returns A random number generator.

Return type numpy.random.Generator

`lenskit.util.`**`init_rng`**(*seed*, **keys*, *propagate=True*)

Initialize the random infrastructure with a seed. This function should generally be called very early in the setup.

Parameters

- **seed** (*int or numpy.random.SeedSequence*) – The random seed to initialize with.
- **keys** – Additional keys, to use as a `spawn_key` on NumPy 1.17. Passed to `derive_seed()`.
- **propagate** (*bool*) – If `True`, initialize other RNG infrastructure. This currently initializes:
  - `np.random.seed()`
  - `random.seed()`

  If `propagate=False`, LensKit is still fully seeded — no component included with LensKit uses any of the global RNGs, they all use RNGs seeded with the specified seed.

Returns The random seed.

`lenskit.util.`**`derivable_rng`**(*spec*, ***, *legacy=False*)

Get a derivable RNG, for use cases where the code needs to be able to reproducibly derive sub-RNGs for different keys, such as user IDs.

Parameters **spec** – Any value supported by the *seed* parameter of *rng()*, in addition to the following values:

- the string `'user'`
- a tuple of the form (*seed*, `'user'`)

Either of these forms will cause the returned function to re-derive new RNGs.

Returns A function taking one (or more) key values, like `derive_seed()`, and returning a random number generator (the type of which is determined by the `legacy` parameter).

Return type function

`lenskit.util.`**`proc_count`**(*core_div=2*, *max_default=None*, *level=0*)

Get the number of desired jobs for multiprocessing operations. This does not affect Numba or MKL multithreading.

This count can come from a number of sources:

- The `LK_NUM_PROCS` environment variable
- The number of CPUs, divided by `core_div` (default 2)

Parameters

- **core_div** (*int or None*) – The divisor to scale down the number of cores; `None` to turn off core-based fallback.
- **max_default** – The maximum number of processes to use if the environment variable is not configured.
- **level** – The process nesting level. 0 is the outermost level of parallelism; subsequent levels control nesting. Levels deeper than 1 are rare, and it isn't expected that callers actually have an accurate idea of the threading nesting, just that they are configuring a child. If the process count is unconfigured, then level 1 will use `core_div`, and deeper levels will use 1.

Returns The number of jobs desired.

Return type int

`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
```

1.20 Random Number Generation

Current best practice for reproducible science in machine learning — including, but not limited to, recommender systems — is to use fixed random seeds so results can be reproduced precisely. This is useful both for reproducing the results themselves and for debugging.

To test for seed sensitivity, the entire experiment can be re-run with a different random seed and the conclusions compared.

LensKit is built to support this experimental design, making consistent use of configurable random number generators throughout its algorithm implementations. When run against NumPy 1.17 or later, it uses the new `numpy.random.Generator` and `numpy.random.SeedSequence` facilities to provide consistent random number generation and initialization. LensKit is compatible with older versions of NumPy, but the RNG reproducibility logic will not fully function, and some functions will not work.

Developers *using* LensKit will be primarily intrested in the *init_rng()* function, so they can initialize LensKit's random seed. LensKit components using randomization also take an `rng` option, usually in their constructor, to set the seed on a per-operation basis; if the script is straightforward and performs LensKit operations in a deterministic order (e.g. does not train multiple models in parallel), initializing the global RNG is sufficient.

Developers writing new LensKit algorithms that use randomization will also need pay attention to the *rng()* function, along with *derivable_rng()* and *derive_seed()* if predictions or recommendations, not just model training, requires random values. Their constructors should take a parameter `rng_spec` to specify the RNG initialization.

1.20.1 Seeds

LensKit random number generation starts from a global root seed, accessible with *get_root_seed()*. This seed can be initialized with *init_rng()*.

`lenskit.util.random.`**`init_rng`**(*seed*, **keys*, *propagate=True*)

Initialize the random infrastructure with a seed. This function should generally be called very early in the setup.

Parameters

- **seed** (*int or numpy.random.SeedSequence*) – The random seed to initialize with.
- **keys** – Additional keys, to use as a `spawn_key` on NumPy 1.17. Passed to *derive_seed()*.
- **propagate** (*bool*) – If `True`, initialize other RNG infrastructure. This currently initializes:
  - `np.random.seed()`
  - `random.seed()`

  If `propagate=False`, LensKit is still fully seeded — no component included with LensKit uses any of the global RNGs, they all use RNGs seeded with the specified seed.

Returns The random seed.

`lenskit.util.random.`**`derive_seed`**(**keys*, *base=None*)

Derive a seed from the root seed, optionally with additional seed keys.

Parameters

- **keys** (*list of int or str*) – Additional components to add to the spawn key for reproducible derivation. If unspecified, the seed's internal counter is incremented.
- **base** (*numpy.random.SeedSequence*) – The base seed to use. If `None`, uses the root seed.

`lenskit.util.random.`**`get_root_seed`**()

Get the root seed.

Returns The LensKit root seed.

Return type numpy.random.SeedSequence

1.20.2 Random Number Generators

These functions create actual RNGs from the LensKit global seed or a user-provided seed. They can produce both new-style `numpy.random.Generator` RNGs and legacy `numpy.random.mtrand.RandomState`; the latter is needed because some libraries, such as Pandas and scikit-learn, do not yet know what to do with a new-style RNG.

`lenskit.util.random.`**`rng`**(*spec=None*, ***, *legacy=False*)

Get a random number generator. This is similar to `sklearn.utils.check_random_seed()`, but it usually returns a `numpy.random.Generator` instead.

Parameters

- **spec** – The spec for this RNG. Can be any of the following types:
  - `int`
  - `None`
  - `numpy.random.SeedSequence`
  - `numpy.random.mtrand.RandomState`
  - `numpy.random.Generator`
- **legacy** (*bool*) – If `True`, return `numpy.random.mtrand.RandomState` instead of a new-style `numpy.random.Generator`.

Returns A random number generator.

Return type numpy.random.Generator

`lenskit.util.random.`**derivable_rng**(*spec*, *, *legacy=False*)

Get a derivable RNG, for use cases where the code needs to be able to reproducibly derive sub-RNGs for different keys, such as user IDs.

Parameters **spec** – Any value supported by the *seed* parameter of *rng()*, in addition to the following values:

- the string `'user'`
- a tuple of the form (*seed*, `'user'`)

Either of these forms will cause the returned function to re-derive new RNGs.

Returns A function taking one (or more) key values, like *derive_seed()*, and returning a random number generator (the type of which is determined by the `legacy` parameter).

Return type function

1.21 LensKit Internals

These modules are primarily for internal infrastructural support in Lenskit. Neither LensKit users nor algorithm developers are likely to need to use this code directly.

1.21.1 Model Sharing

The *lenskit.sharing* module provides utilities for managing models and sharing them between processes, particularly for the multiprocessing in *lenskit.batch*.

Sharing Mode

The only piece **algorithm developers** usually need to directly handle is the concept of 'sharing mode' when implementing custom pickling logic. To save space, it is reasonable to exclude intermediate data structures, such as caches or inverse indexes, from the pickled representation of an algorithm, and reconstruct them when the model is loaded.

However, LensKit's multi-process sharing *also* uses pickling to capture the object state while using shared memory for `numpy.ndarray` objects. In these cases, the structures should be pickled, so they can be shared between model instances.

To support this, we have the concept of *sharing mode*. Code that excludes objects when pickling should call *in_share_context()* to determine if that exclusion should actually happen.

lenskit.sharing.**in_share_context**()
Query whether sharing mode is active. If `True`, we are currently in a *sharing_mode()* context, which means model pickling will be used for cross-process sharing.

lenskit.sharing.**sharing_mode**(**args*, ***kwds*)
Context manager to tell models that pickling will be used for cross-process sharing, not model persistence.

Persistence API

These functions are used for internal LensKit infrastructure code to persist models into shared memory for parallel processing.

lenskit.sharing.**persist**(*model*, *, *method=None*)
Persist a model for cross-process sharing.

This will return a persiste dmodel that can be used to reconstruct the model in a worker process (using `reconstruct()`).

If no method is provided, this function automatically selects a model persistence strategy from the the following, in order:

1. If *LK_TEMP_DIR* is set, use `binpickle` in shareable mode to save the object into the LensKit temporary directory.
2. If `multiprocessing.shared_memory` is available, use `pickle` to save the model, placing the buffers into shared memory blocks.
3. Otherwise, use `binpickle` in shareable mode to save the object into the system temporary directory.

Parameters

- **model** (*obj*) – The model to persist.
- **method** (*str or None*) – The method to use. Can be one of `binpickle` or `shm`.

Returns The persisted object.

Return type *PersistedModel*

class lenskit.sharing.**PersistedModel**
Bases: `abc.ABC`

A persisted model for inter-process model sharing.

These objects can be pickled for transmission to a worker process.

Note: Subclasses need to override the pickling protocol to implement the proper pickling implementation.

---

abstract get()
Get the persisted model, reconstructing it if necessary.

abstract close()
Release the persisted model resources. Should only be called in the parent process (will do nothing in a child process).

transfer()
Mark an object for ownership transfer. This object, when pickled, will unpickle into an owning model that frees resources when closed. Used to transfer ownership of shared memory resources from child processes to parent processes. Such an object should only be unpickled once.

The default implementation sets the `is_owner` attribute to `'transfer'`.

Returns `self` (for convenience)

1.21.2 Parallel Execution

LensKit uses `concurrent.futures.ProcessPoolExecutor` to paralellize batch operations (see *lenskit.batch*).

The basic idea of this API is to create an *invoker* that has a model and a function, and then passing lists of argument sets to the function:

```
with invoker(model, func):
    results = list(func.map(args))
```

The model is persisted into shared memory to be used by the worker processes.

Parallel Model Ops

`lenskit.util.parallel.`**`invoker`**(*model*, *func*, *n_jobs=None*, ***, *persist_method=None*)

Get an appropriate invoker for performing oeprations on `model`.

Parameters

- **`model`** (*obj*) – The model object on which to perform operations.
- **`func`** (*function*) – The function to call. The function must be pickleable.
- **`n_jobs`** (*int or None*) – The number of processes to use for parallel operations. If `None`, will call *proc_count()* with a maximum default process count of 4.
- **`persist_method`** (*str or None*) – The persistence method to use. Passed as `method` to *lenskit.sharing.persist()*.

Returns An invoker to perform operations on the model.

Return type *ModelOpInvoker*

`lenskit.util.parallel.`**`proc_count`**(*core_div=2*, *max_default=None*, *level=0*)

Get the number of desired jobs for multiprocessing operations. This does not affect Numba or MKL multithreading.

This count can come from a number of sources:

- The `LK_NUM_PROCS` environment variable
- The number of CPUs, divided by `core_div` (default 2)

Parameters

- **`core_div`** (*int or None*) – The divisor to scale down the number of cores; `None` to turn off core-based fallback.
- **`max_default`** – The maximum number of processes to use if the environment variable is not configured.
- **`level`** – The process nesting level. 0 is the outermost level of parallelism; subsequent levels control nesting. Levels deeper than 1 are rare, and it isn't expected that callers actually have an accurate idea of the threading nesting, just that they are configuring a child. If the process count is unconfigured, then level 1 will use `core_div`, and deeper levels will use 1.

Returns The number of jobs desired.

Return type int

class lenskit.util.parallel.**ModelOpInvoker**

Bases: abc.ABC

Interface for invoking operations on a model, possibly in parallel. The operation invoker is configured with a model and a function to apply, and applies that function to the arguments supplied in *map*. Child process invokers also route logging messages to the parent process, so logging works even with multiprocessing.

An invoker is a context manager that calls `shutdown()` when exited.

abstract map(**iterables*)

Apply the configured function to the model and iterables. This is like *map()*, except it supplies the invoker's model as the first object to `func`.

Parameters **iterables** – Iterables of arguments to provide to the function.

Returns An iterable of the results.

Return type iterable

Single Process Isolation

We also have a single-process isolation function that runs a function in a subprocess.

lenskit.util.parallel.**run_sp**(*func*, **args*, ***kwargs*)

Run a function in a subprocess and return its value. This is for achieving subprocess isolation, not parallelism. The subprocess is configured so things like logging work correctly.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. IIS 17-51278. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

BIBLIOGRAPHY

[LKPY] Michael D. Ekstrand. 2018. The LKPY Package for Recommender Systems Experiments: Next-Generation Tools and Lessons Learned from the LensKit Project. *Computer Science Faculty Publications and Presentations* 147. Boise State University. DOI:10.18122/cs_facpubs/147/boisestate. arXiv:1809.03125 [cs.IR].

[HK2015] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems* (TiiS) **5**, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

[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].

[ZWSP2008] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In +Algorithmic Aspects in Information and **Management**_, LNCS 5034, 337–348. DOI 10.1007/978-3-540-68880-8_32.

[TPT2011] Gábor Takács, István Pilászy, and Domonkos Tikk. 2011. Applications of the Conjugate Gradient Method for Implicit Feedback Collaborative Filtering.

[HKV2008] Y. Hu, Y. Koren, and C. Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In _Proceedings of the 2008 Eighth IEEE International Conference on Data **Mining**_, 263–272. DOI 10.1109/ICDM.2008.22

[TPT2011] Gábor Takács, István Pilászy, and Domonkos Tikk. 2011. Applications of the Conjugate Gradient Method for Implicit Feedback Collaborative Filtering.

[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

E

F

G

H

I

L

M

N

P

R

S

T

U

V