LensKit Documentation

Release 0.8.4

Michael D. Ekstrand

Jan 09, 2020

CONTENTS:

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

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

CHAPTER

ONE

INSTALLATION

To install the current release with Anaconda (recommended):

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

Or you can use `pip`:

```
pip install lenskit
```

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

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

Then see Getting Started.

CHAPTER

TWO

RESOURCES

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

2.1 Getting Started

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

2.1.1 Setup

We first import the LensKit components we need:

[1]:
```
from lenskit.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
```

2.1.2 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

2.1.3 Defining Algorithms

Let's set up two algorithms:

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

2.1.4 Running the Evaluation

In LensKit, our evaluation proceeds in 2 steps:

1. Generate recommendations
2. Measure them

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

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

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

Note 2: our algorithms do not necessarily implement the `Recommend` 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
2    ALS        0.148335
     ItemItem   0.081890
3    ALS        0.026615
```

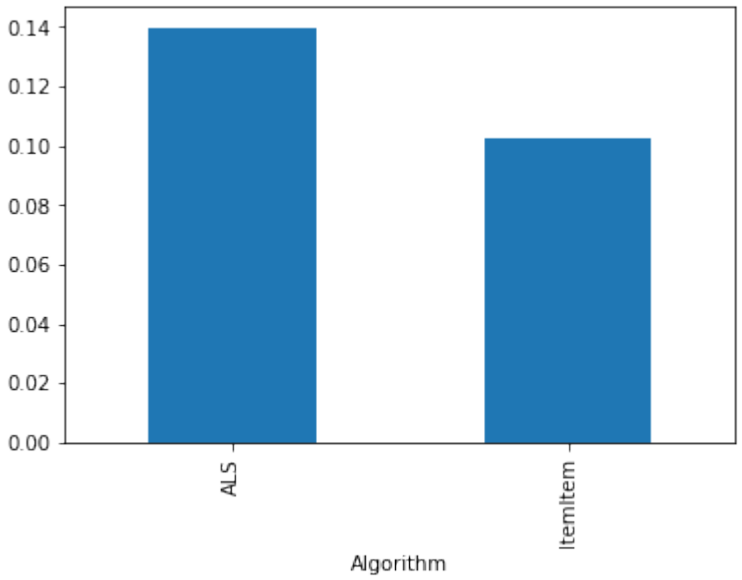
Now we have nDCG values!

[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>
```



[ ]:

2.2 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()`.

2.2.1 Base Algorithm

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

The *`Algorithm`* interface defines common methods.

class lenskit.algorithms.**Algorithm**

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

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

2.2.2 Recommendation

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

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

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

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

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.

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

2.2.3 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.algorithms.**CandidateSelector**
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()*.

2.2.4 Rating Prediction

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

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

Parameters

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

Returns The predicted scores for each user-item pair.

Return type pandas.Series

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

2.3 Crossfold preparation

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

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

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

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

2.3.1 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*)

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

Parameters

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

Return type iterator

Returns an iterator of train-test pairs

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

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

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

```
>>> ratings = util.load_ml_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.

Returns An iterator of train-test pairs.

Return type iterator

2.3.2 User-based splitting

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

The algorithm used by each is as follows:

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

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

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

Parameters

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

Return type iterator

Returns an iterator of train-test pairs

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

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

Parameters

- **data** (*pandas.DataFrame*) – 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.

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

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

lenskit.crossfold.**SampleFrac**(*frac*)
Randomly select a fraction of test rows per user/item.

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

lenskit.crossfold.**LastN**(*n*, *col='timestamp'*)
Select a fixed number of test rows per user/item, based on ordering by a column.

Parameters

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

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

Parameters

- **frac** – the fraction of items to select for testing.
- **col** – The column to sort by.

2.3.3 Utility Classes

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

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

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

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

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

property test
Test data for this pair.

property train
Train data for this pair.

2.4 Batch-Running Recommenders

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

2.4.1 Recommendation

lenskit.batch.**recommend**(*algo*, *users*, *n*, *candidates=None*, ***, *n_jobs=None*, *dask_result=False*, ***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 as `n_jobs` to **:cls:`joblib.Parallel`**. The default, `None`, will make the process sequential _unless_ called inside the `joblib.parallel_backend()` context manager.

  Note: `nprocs` is accepted as a deprecated alias.

- **dask_result** (*bool*) – Whether to return a Dask data frame instead of a Pandas one.

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

2.4.2 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
>>> ratings = util.load_ml_ratings()
>>> bias = Bias()
>>> bias.fit(ratings[:-1000])
```

(continues on next page)

(continued from previous page)

```
<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 as `n_jobs` to **:cls:`joblib.Parallel`**. The default, `None`, will make the process sequential _unless_ called inside the `joblib.parallel_backend()` context manager.

  Note: `nprocs` is accepted as a deprecated alias.

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

Return type pandas.DataFrame

2.4.3 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.util import load_ml_ratings
from lenskit import topn
import pandas as pd
```

Generate the train-test pairs:

```
pairs = list(partition_users(load_ml_ratings(), 5, SampleN(5)))
```

Set up and run the `MultiEval` experiment:

```
eval = MultiEval('my-eval', recommend=20)
eval.add_datasets(pairs, name='ML-Small')
```

(continues on next page)

(continued from previous page)

```
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()
```

Now that the experiment is run, we can read its outputs.

First the run metadata:

```
runs = pd.read_csv('my-eval/runs.csv')
runs.set_index('RunId', inplace=True)
runs.head()
```

Then the recommendations:

```
recs = pd.read_parquet('my-eval/recommendations.parquet')
recs.head()
```

```
D:Anaconda3libsite-packagespyarrowpandas_compat.py:698: FutureWarning: .
↪labels was deprecated in version 0.24.0. Use .codes instead.
  labels = getattr(columns, 'labels', None) or [
D:Anaconda3libsite-packagespyarrowpandas_compat.py:725: FutureWarning: the␣
↪'labels' keyword is deprecated, use 'codes' instead
  return pd.MultiIndex(levels=new_levels, labels=labels, names=columns.names)
D:Anaconda3libsite-packagespyarrowpandas_compat.py:742: FutureWarning: .
↪labels was deprecated in version 0.24.0. Use .codes instead.
  labels, = index.labels
```

In order to evaluate the recommendation list, we need to build a combined set of truth data. Since this is a disjoint partition of users over a single data set, we can just concatenate the individual test frames:

```
truth = pd.concat((p.test for p in pairs), ignore_index=True)
```

Now we can set up an analysis and compute the results.

```
rla = topn.RecListAnalysis()
rla.add_metric(topn.ndcg)
ndcg = rla.compute(recs, truth)
ndcg.head()
```

Next, we need to combine this with our run data, so that we know what algorithms and configurations we are evaluating:

```
ndcg = ndcg.join(runs[['AlgoClass', 'features']], on='RunId')
ndcg.head()
```

The Popular algorithm has NaN feature count, which `groupby` doesn't like; let's fill those in.

```
ndcg.loc[ndcg['AlgoClass'] == 'Popular', 'features'] = 0
```

And finally, we can compute the overall average performance for each algorithm configuration:

```
ndcg.groupby(['AlgoClass', 'features'])['ndcg'].mean()
```

```
AlgoClass  features
BiasedMF   20.0        0.015960
```

(continues on next page)

(continued from previous page)

```
            30.0        0.022558
            40.0        0.025901
            50.0        0.028949
Popular     0.0         0.091814
Name: ndcg, dtype: float64
```

Multi-Eval Class Reference

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

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; if `'joblib'`, they are pickled with `joblib.dump()` to uncompressed `.jlpkl` 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.

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

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

run(*runs=None*, *, *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.

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

2.5 Evaluating Recommender Output

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

We generally recommend using Jupyter notebooks for evaluation.

2.5.1 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)
```

2.5.2 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*)

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. 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

2.5.3 Loading Outputs

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

There are several files:

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

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

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

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

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

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

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

2.6 Data Set Utilities

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`).

2.6.1 MovieLens Data Sets

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

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

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 links

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

```
>>> mlsmall = MovieLens('ml-latest-small')
>>> mlsmall.links
         imdbId  tmdbId
item
1        114709     862
2        113497    8844
3        113228   15602
4        114885   31357
5        113041   11862
...
[9125 rows x 2 columns]
```

property movies

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

```
>>> mlsmall = MovieLens('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 ratings

The rating table.

```
>>> mlsmall = MovieLens('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 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.

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

```
>>> mlsmall = MovieLens('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]
```

class lenskit.datasets.**ML100K**(*path='data/ml-100k'*)
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 movies
Return the user data (from `u.user`).

```
>>> ml = ML100K('ml-100k')
>>> 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]
```

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

```
>>> ml = ML100K('ml-100k')
>>> 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-100k')
>>> 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]
```

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

MovieLens 1M data set.

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

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]
```

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

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]
```

2.7 Algorithms

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

2.7.1 Basic and Utility Algorithms

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

Personalized Mean Rating Prediction

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

Bases: *lenskit.algorithms.Predictor*

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

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

where 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_

The global mean rating.

Type double

item_offsets_

The item offsets (b_i values)

Type pandas.Series

user_offsets_

The item offsets (b_u values)

Type pandas.Series

fit(*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*

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

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*.

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

Random Item Recommendation

The *Random* algorithm implements random-item recommendation.

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

A random-item recommender.

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

Type *CandidateSelector*

random_state
Seed or random state for generating recommendations.

Type int or numpy.random.RandomState

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

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.

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

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

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_

All known items.

Type pandas.Index

users_

All known users.

Type pandas.Index

user_items_

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

Type *CSR*

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.

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.

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

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

2.7.2 k-NN Collaborative Filtering

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

Item-based k-NN

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

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

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_
the index of item IDs.

Type pandas.Index

item_means_
the mean rating for each known item.

Type numpy.ndarray

item_counts_
the number of saved neighbors for each item.

Type numpy.ndarray

sim_matrix_
the similarity matrix.

Type *matrix.CSR*

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

Type pandas.Index

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

Type *matrix.CSR*

fit (*ratings*, ***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

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_
User index.

Type pandas.Index

item_index_
Item index.

Type pandas.Index

user_means_
User mean ratings.

Type numpy.ndarray

rating_matrix_
Normalized user-item rating matrix.

Type *matrix.CSR*

transpose_matrix_
Transposed un-normalized rating matrix.

Type *matrix.CSR*

fit(*ratings*, ***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

2.7.3 Classic Matrix Factorization

LKPY provides classical matrix factorization implementations.

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**

Common predictor for matrix factorization.

user_index_

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

Type pandas.Index

item_index_

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

Type pandas.Index

user_features_

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

Type numpy.ndarray

item_features_

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

Type numpy.ndarray

lookup_items(*items*)

Look up the indices for a set of items.

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

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

Return type numpy.ndarray

lookup_user(*user*)

Look up the index for a user.

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

Returns the user index.

Return type int

property n_features

The number of features.

property n_items

The number of items.

property n_users

The number of users.

score(*user*, *items*)

Score a set of items for a user. User and item parameters must be indices into the matrices.

Parameters

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

Returns the scores for the items.

Return type numpy.ndarray

class lenskit.algorithms.mf_common.**BiasMFPredictor**

Common model for biased matrix factorization.

user_index_

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

Type pandas.Index

item_index_

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

Type pandas.Index

global_bias_

The global bias term.

Type double

user_bias_

The user bias terms.

Type numpy.ndarray

item_bias_

The item bias terms.

Type numpy.ndarray

user_features_

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

Type numpy.ndarray

item_features_

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

Type numpy.ndarray

score(*user*, *items*, *raw=False*)

Score a set of items for a user. User and item parameters must be indices into the matrices.

Parameters

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

Returns the scores for the items.

Return type numpy.ndarray

Alternating Least Squares

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

class lenskit.algorithms.als.**BiasedMF**(*features*, ***, *iterations=20*, *reg=0.1*, *damping=5*, *bias=True*, *method='cd'*, *rand=<built-in method randn of numpy.random.mtrand.RandomState object>*, *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** (*function*) – RNG function compatible with **:fun:`numpy.random.randn`** for initializing matrices.
- **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).

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'*, *rand=<built-in method randn of numpy.random.mtrand.RandomState object>*, *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]).
- **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

FunkSVD

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

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

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.

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

2.7.4 Hierarchical Poisson Factorization

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

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

Parameters

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

fit(*ratings*, ***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

2.7.5 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*)
LensKit interface to implicit.als.

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

2.7.6 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.

2.7.7 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.

2.7.8 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.

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_**.

2.7.9 References

2.8 Utility Functions

2.8.1 Matrix Utilities

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

Building Ratings Matrices

lenskit.matrix.**sparse_ratings**(*ratings*, *scipy=False*)

Convert a rating table to a sparse matrix of ratings.

Parameters

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

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

Return type *RatingMatrix*

class lenskit.matrix.**RatingMatrix**

A rating matrix with associated indices.

matrix

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

Type *CSR* or scipy.sparse.csr_matrix

users

mapping from user IDs to row numbers.

Type pandas.Index

items

mapping from item IDs to column numbers.

Type pandas.Index

Compressed Sparse Row Matrices

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

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

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

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

Type *_CSR*

nrows
the number of rows.

Type int

ncols
the number of columns.

Type int

nnz
the number of entries.

Type int

rowptrs
the row pointers.

Type numpy.ndarray

colinds
the column indices.

Type numpy.ndarray

values
the values

Type numpy.ndarray

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

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*

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*

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

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

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_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

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

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

Note: This method is not available from Numba.

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

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

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*

class lenskit.matrix.**_CSR**(*nrows*, *ncols*, *nnz*, *ptrs*, *inds*, *vals*)
Internal implementation class for *CSR*. If you work with CSRs from Numba, you will use 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.

2.8.2 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**()

lenskit.math.solve.**_dtrsv**()

2.8.3 Miscellaneous

Miscellaneous utility functions.

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

  This function is somewhat derived from the SciKit one.

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

lenskit.util.**cur_memory**()
: Get the current memory use for this process

lenskit.util.**max_memory**()
: Get the maximum memory use for this process

2.9 Errors and Diagnostics

2.9.1 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.

2.9.2 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**
: Warning raised for detectable problems with input data.

2.10 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.

2.10.1 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.

2.10.2 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.

2.10.3 Memory Map Friendliness

LensKit uses **:py:cls:`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 **:py:cls:`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 **:py:cls:`lenskit.algorithms.basic.UnratedItemSelector`**, we do this by storing user and item indexes along with a **:py:cls:`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.

2.11 Release Notes

2.11.1 0.8.4

This release cleans up dependency problems to make it easier to reliably install LensKit. We remove some unused utility code that had compatibility problems.

- Remove `CSR.sort_values` - we were no longer using this function, and it failed to compile with Numba 0.46.
- Change dependency versions

2.11.2 0.8.3

- Deprecated `lenskit.util.write_parquet`
- Made `MultiEval` use directories of parquet files, instead of appending to a single file, for writing multiple results to improve performance without fastparquet.
- Updated Numba dependency to fix incompatibilities (#129).

2.11.3 0.8.0

See the GitHub milestone for full change list.

Infrastructure Updates

- Dropped support for Python 3.5
- Removed `*args` from `Algorithm.fit`, so additional data must be provided via keyword arguments
- Made `Algorithm.fit` implementations consistently take `**kwargs` for hybrid flexibility

Algorithm Updates

- Substantial performance and stability improvements to item-item
- Added a coordinate descent solver to explicit-feedback ALS and made it the default. The old LU-based solver is still available with `method='lu'`.
- Added a conjugate gradient solver to implicit-feedback ALS and made it the default.
- Added a random recommender

2.11.4 0.7.0

See the GitHub milestone for full change list.

- Use Joblib for parallelism in batch routines.
- `nprocs` arguments are renamed to `n_jobs` for consistency with Joblib.
- Removed `parallel` option on `MultiEval` algorithms, as it was unused.
- Made `MultiEval` default to using each recommender's default candidate set, and adapt algorithms to recommenders prior to evaluation.

- Make `MultiEval` require named arguments for most things.
- Add support to `MultiEval` to save the fit models.
- `RecListAnalysis` can optionally ensure all test users are returned, even if they lack recommendation lists.
- Performance improvements to algorithms and evaluation.

2.11.5 0.6.1

See the GitHub milestone for full change list.

- Fix inconsistency in both code and docs for recommend list sizes for top-*N* evaluation.
- Fix user-user to correctly use `sum` aggregate.
- Improve performance and documentation

2.11.6 0.6.0

See the GitHub milestone for a summary of what's happening!

- The `save` and `load` methods on algorithms have been removed. Just pickle fitted models to save their data. This is what SciKit does, we see no need to deviate.
- The APIs and model structures for top-*N* recommendation is reworked to enable algorithms to produce recommendations more automatically. The `Recommender` interfaces now take a `CandidateSelector` to determine default candidates, so client code does not need to compute candidates on their own. One effect of this is that the `batch.recommend` function no longer requires a candidate selector, and there can be problems if you call `Recommender.adapt` before fitting a model.
- Top-*N* evaluation has been completely revamped to make it easier to correctly implement and run evaluation metrics. Batch recommend no longer attaches ratings to recommendations. See Top-N evaluation for details.
- Batch recommend & predict functions now take `nprocs` as a keyword-only argument.
- Several bug fixes and testing improvements.

Internal Changes

These changes should not affect you if you are only consuming LensKit's algorithm and evaluation capabilities.

- Rewrite the `CSR` class to be more ergonomic from Python, at the expense of making the NumPy jitclass indirect. It is available in the `.N` attribute. Big improvement: it is now picklable.

2.11.7 0.5.0

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

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

We also have some new capabilities:

- Ben Frederickson's Implicit library

2.11.8 0.3.0

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

2.11.9 0.2.0

A lot of fixes to get ready for RecSys.

2.11.10 0.1.0

Hello, world!

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

FOUR

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

[ML] 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

[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.

[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].

PYTHON MODULE INDEX

l

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V