idly Documentation

Release 1

Nicolas Hug

May 12, 2018

User Guide

idly a collection of interpretable algorithms realized via DNN architectures

If you're new to idly, we invite you to take a look at the *Getting Started* guide, where you'll find a series of tutorials illustrating all you can do with idly. You can also check out the *FAQ* for many use-case example.

Any kind of feedback/criticism would be greatly appreciated (software design, documentation, improvement ideas, spelling mistakes, etc…). Please feel free to contribute and send pull requests (see GitHub page)!

CHAPTER 1

Getting Started

1.1 Basic usage

1.1.1 Automatic cross-validation

Surprise has a set of built-in *algorithms* and *datasets* for you to play with. In its simplest form, it only takes a few lines of code to run a cross-validation procedure:

Listing 1.1: From file `examples/basic_usage.py`

```
from surprise import SVD
from surprise import Dataset
from surprise.model_selection import cross_validate


# Load the movielens-100k dataset (download it if needed),
data = Dataset.load_builtin('ml-100k')

# We'll use the famous SVD algorithm.
algo = SVD()

# Run 5-fold cross-validation and print results
cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5,␣
↪verbose=True)
```

The result should be as follows (actual values may vary due to randomization):

```
Evaluating RMSE, MAE of algorithm SVD on 5 split(s).

            Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE        0.9311  0.9370  0.9320  0.9317  0.9391  0.9342  0.0032
MAE         0.7350  0.7375  0.7341  0.7342  0.7375  0.7357  0.0015
Fit time    6.53    7.11    7.23    7.15    3.99    6.40    1.23
Test time   0.26    0.26    0.25    0.15    0.13    0.21    0.06
```

The `load_builtin()` method will offer to download the movielens-100k dataset if it has not already been downloaded, and it will save it in the `.idly_data` folder in your home directory (you can also choose to save it *somewhere else*).

We are here using the well-known `SVD` algorithm, but many other algorithms are available. See *Using prediction algorithms* for more details.

The `cross_validate()` function runs a cross-validation procedure according to the `cv` argument, and computes some `accuracy` measures. We are here using a classical 5-fold cross-validation, but fancier iterators can be used (see *here*).

1.1.2 Train-test split and the fit() method

If you don't want to run a full cross-validation procedure, you can use the `train_test_split()` to sample a trainset and a testset with given sizes, and use the `accuracy metric` of your chosing. You'll need to use the `fit()` method which will train the algorithm on the trainset, and the `test()` method which will return the predictions made from the testset:

Listing 1.2: From file `examples/train_test_split.py`

```
from surprise import SVD
from surprise import Dataset
from surprise import accuracy
from surprise.model_selection import train_test_split

# Load the movielens-100k dataset (download it if needed),
data = Dataset.load_builtin('ml-100k')

# sample random trainset and testset
# test set is made of 25% of the ratings.
trainset, testset = train_test_split(data, test_size=.25)

# We'll use the famous SVD algorithm.
algo = SVD()

# Train the algorithm on the trainset, and predict ratings for the␣
↪testset
```

```
algo.fit(trainset)
predictions = algo.test(testset)

# Then compute RMSE
accuracy.rmse(predictions)
```

Result:

```
RMSE: 0.9411
```

Note that you can train and test an algorithm with the following one-line:

```
predictions = algo.fit(trainset).test(testset)
```

In some cases, your trainset and testset are already defined by some files. Please refer to *this section* to handle such cases.

1.1.3 Train on a whole trainset and the predict() method

Obviously, we could also simply fit our algorithm to the whole dataset, rather than running cross-validation. This can be done by using the `build_full_trainset()` method which will build a `trainset` object:

Listing 1.3: From file `examples/predict_ratings.py`

```
from surprise import KNNBasic
from surprise import Dataset

# Load the movielens-100k dataset
data = Dataset.load_builtin('ml-100k')

# Retrieve the trainset.
trainset = data.build_full_trainset()

# Build an algorithm, and train it.
algo = KNNBasic()
algo.fit(trainset)
```

We can now predict ratings by directly calling the `predict()` method. Let's say you're interested in user 196 and item 302 (make sure they're in the trainset!), and you know that the true rating $r_{ui} = 4$:

Listing 1.4: From file `examples/predict_ratings.py`

```
uid = str(196)  # raw user id (as in the ratings file). They are␣
↪**strings**!
```

```
iid = str(302)  # raw item id (as in the ratings file). They are␣
↪**strings**!

# get a prediction for specific users and items.
pred = algo.predict(uid, iid, r_ui=4, verbose=True)
```

The result should be:

```
user: 196        item: 302        r_ui = 4.00   est = 4.06   {'actual_k
↪': 40, 'was_impossible': False}
```

Note: The `predict()` uses **raw** ids (please read *this* about raw and inner ids). As the dataset we have used has been read from a file, the raw ids are strings (even if they represent numbers).

We have so far used a built-in dataset, but you can of course use your own. This is explained in the next section.

1.2 Use a custom dataset

Surprise has a set of builtin *datasets*, but you can of course use a custom dataset. Loading a rating dataset can be done either from a file (e.g. a csv file), or from a pandas dataframe. Either way, you will need to define a `Reader` object for idly to be able to parse the file or the dataframe.

- To load a dataset from a file (e.g. a csv file), you will need the `load_from_file()` method:

Listing 1.5: From file `examples/load_custom_dataset.py`

```
from surprise import BaselineOnly
from surprise import Dataset
from surprise import Reader
from surprise.model_selection import cross_validate

# path to dataset file
file_path = os.path.expanduser('~/.surprise_data/ml-100k/ml-100k/u.
↪data')

# As we're loading a custom dataset, we need to define a reader.␣
↪In the
# movielens-100k dataset, each line has the following format:
# 'user item rating timestamp', separated by '\t' characters.
reader = Reader(line_format='user item rating timestamp', sep='\t')

data = Dataset.load_from_file(file_path, reader=reader)
```

```
# We can now use this dataset as we please, e.g. calling cross_
↪validate
cross_validate(BaselineOnly(), data, verbose=True)
```

For more details about readers and how to use them, see the `Reader class` documentation.

Note: As you already know from the previous section, the Movielens-100k dataset is built-in so a much quicker way to load the dataset is to do `data = Dataset.load_builtin('ml-100k')`. We will of course ignore this here.

- To load a dataset from a pandas dataframe, you will need the `load_from_df()` method. You will also need a `Reader` object, but only the `rating_scale` parameter must be specified. The dataframe must have three columns, corresponding to the user (raw) ids, the item (raw) ids, and the ratings in this order. Each row thus corresponds to a given rating. This is not restrictive as you can reorder the columns of your dataframe easily.

Listing 1.6: From file `examples/load_from_dataframe.py`

```
import pandas as pd

from surprise import NormalPredictor
from surprise import Dataset
from surprise import Reader
from surprise.model_selection import cross_validate


# Creation of the dataframe. Column names are irrelevant.
ratings_dict = {'itemID': [1, 1, 1, 2, 2],
                'userID': [9, 32, 2, 45, 'user_foo'],
                'rating': [3, 2, 4, 3, 1]}
df = pd.DataFrame(ratings_dict)

# A reader is still needed but only the rating_scale param is␣
↪requiered.
reader = Reader(rating_scale=(1, 5))

# The columns must correspond to user id, item id and ratings (in␣
↪that order).
data = Dataset.load_from_df(df[['userID', 'itemID', 'rating']],␣
↪reader)

# We can now use this dataset as we please, e.g. calling cross_
↪validate
```

```
cross_validate(NormalPredictor(), data, cv=2)
```

The dataframe initially looks like this:

```
      itemID  rating    userID
0          1       3         9
1          1       2        32
2          1       4         2
3          2       3        45
4          2       1  user_foo
```

1.3 Use cross-validation iterators

For cross-validation, we can use the `cross_validate()` function that does all the hard work for us. But for a better control, we can also instanciate a cross-validation iterator, and make predictions over each split using the `split()` method of the iterator, and the `test()` method of the algorithm. Here is an example where we use a classical K-fold cross-validation procedure with 3 splits:

Listing 1.7: From file `examples/use_cross_validation_iterators.py`

```
from surprise import SVD
from surprise import Dataset
from surprise import accuracy
from surprise.model_selection import KFold

# Load the movielens-100k dataset
data = Dataset.load_builtin('ml-100k')

# define a cross-validation iterator
kf = KFold(n_splits=3)

algo = SVD()

for trainset, testset in kf.split(data):

    # train and test algorithm.
    algo.fit(trainset)
    predictions = algo.test(testset)

    # Compute and print Root Mean Squared Error
    accuracy.rmse(predictions, verbose=True)
```

Result could be, e.g.:

```
RMSE: 0.9374
RMSE: 0.9476
RMSE: 0.9478
```

Other cross-validation iterator can be used, like LeaveOneOut or ShuffleSplit. See all the available iterators *here*. The design of idly's cross-validation tools is heavily inspired from the excellent scikit-learn API.

A special case of cross-validation is when the folds are already predefined by some files. For instance, the movielens-100K dataset already provides 5 train and test files (u1.base, u1.test ... u5.base, u5.test). idly can handle this case by using a `idly.model_selection.split.PredefinedKFold` object:

Listing 1.8: From file `examples/load_custom_dataset_predefined_folds.py`

```
from surprise import SVD
from surprise import Dataset
from surprise import Reader
from surprise import accuracy
from surprise.model_selection import PredefinedKFold

# path to dataset folder
files_dir = os.path.expanduser('~/.surprise_data/ml-100k/ml-100k/')

# This time, we'll use the built-in reader.
reader = Reader('ml-100k')

# folds_files is a list of tuples containing file paths:
# [(u1.base, u1.test), (u2.base, u2.test), ... (u5.base, u5.test)]
train_file = files_dir + 'u%d.base'
test_file = files_dir + 'u%d.test'
folds_files = [(train_file % i, test_file % i) for i in (1, 2, 3, 4,␣
↪5)]

data = Dataset.load_from_folds(folds_files, reader=reader)
pkf = PredefinedKFold()

algo = SVD()

for trainset, testset in pkf.split(data):

    # train and test algorithm.
    algo.fit(trainset)
    predictions = algo.test(testset)

    # Compute and print Root Mean Squared Error
```

```
    accuracy.rmse(predictions, verbose=True)
```

Of course, nothing prevents you from only loading a single file for training and a single file for testing. However, the `folds_files` parameter still needs to be a `list`.

1.4 Tune algorithm parameters with GridSearchCV

The `cross_validate()` function reports accuracy metric over a cross-validation procedure for a given set of parameters. If you want to know which parameter combination yields the best results, the `GridSearchCV` class comes to the rescue. Given a `dict` of parameters, this class exhaustively tries all the combinations of parameters and reports the best parameters for any accuracy measure (averaged over the different splits). It is heavily inspired from scikit-learn's GridSearchCV.

Here is an example where we try different values for parameters `n_epochs`, `lr_all` and `reg_all` of the `SVD` algorithm.

Listing 1.9: From file `examples/grid_search_usage.py`

```
from surprise import SVD
from surprise import Dataset
from surprise.model_selection import GridSearchCV

# Use movielens-100K
data = Dataset.load_builtin('ml-100k')

param_grid = {'n_epochs': [5, 10], 'lr_all': [0.002, 0.005],
              'reg_all': [0.4, 0.6]}
gs = GridSearchCV(SVD, param_grid, measures=['rmse', 'mae'], cv=3)

gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])

# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

Result:

```
0.961300130118
{'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.4}
```

We are here evaluating the average RMSE and MAE over a 3-fold cross-validation procedure, but any *cross-validation iterator* can used.

Once `fit()` has been called, the `best_estimator` attribute gives us an algorithm instance with the optimal set of parameters, which can be used how we please:

Listing 1.10: From file `examples/grid_search_usage.py`

```
# We can now use the algorithm that yields the best rmse:
algo = gs.best_estimator['rmse']
algo.fit(data.build_full_trainset())
```

Note: Dictionary parameters such as `bsl_options` and `sim_options` require particular treatment. See usage example below:

```
param_grid = {'k': [10, 20],
              'sim_options': {'name': ['msd', 'cosine'],
                              'min_support': [1, 5],
                              'user_based': [False]}
              }
```

Naturally, both can be combined, for example for the `KNNBaseline` algorithm:

```
param_grid = {'bsl_options': {'method': ['als', 'sgd'],
                              'reg': [1, 2]},
              'k': [2, 3],
              'sim_options': {'name': ['msd', 'cosine'],
                              'min_support': [1, 5],
                              'user_based': [False]}
              }
```

For further analysis, the `cv_results` attribute has all the needed information and can be imported in a pandas dataframe:

Listing 1.11: From file `examples/grid_search_usage.py`

```
results_df = pd.DataFrame.from_dict(gs.cv_results)
```

In our example, the `cv_results` attribute looks like this (floats are formatted):

```
'split0_test_rmse': [1.0, 1.0, 0.97, 0.98, 0.98, 0.99, 0.96, 0.97]
'split1_test_rmse': [1.0, 1.0, 0.97, 0.98, 0.98, 0.99, 0.96, 0.97]
'split2_test_rmse': [1.0, 1.0, 0.97, 0.98, 0.98, 0.99, 0.96, 0.97]
'mean_test_rmse':   [1.0, 1.0, 0.97, 0.98, 0.98, 0.99, 0.96, 0.97]
'std_test_rmse':    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
'rank_test_rmse':   [7 8 3 5 4 6 1 2]
'split0_test_mae':  [0.81, 0.82, 0.78, 0.79, 0.79, 0.8, 0.77, 0.79]
'split1_test_mae':  [0.8, 0.81, 0.78, 0.79, 0.78, 0.79, 0.77, 0.78]
'split2_test_mae':  [0.81, 0.81, 0.78, 0.79, 0.78, 0.8, 0.77, 0.78]
```

```
'mean_test_mae':     [0.81, 0.81, 0.78, 0.79, 0.79, 0.8, 0.77, 0.78]
'std_test_mae':      [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
'rank_test_mae':     [7 8 2 5 4 6 1 3]
'mean_fit_time':     [1.53, 1.52, 1.53, 1.53, 3.04, 3.05, 3.06, 3.02]
'std_fit_time':      [0.03, 0.04, 0.0, 0.01, 0.04, 0.01, 0.06, 0.01]
'mean_test_time':    [0.46, 0.45, 0.44, 0.44, 0.47, 0.49, 0.46, 0.34]
'std_test_time':     [0.0, 0.01, 0.01, 0.0, 0.03, 0.06, 0.01, 0.08]
'params':            [{'n_epochs': 5, 'lr_all': 0.002, 'reg_all': 0.4},
↪{'n_epochs': 5, 'lr_all': 0.002, 'reg_all': 0.6}, {'n_epochs': 5,
↪'lr_all': 0.005, 'reg_all': 0.4}, {'n_epochs': 5, 'lr_all': 0.005,
↪'reg_all': 0.6}, {'n_epochs': 10, 'lr_all': 0.002, 'reg_all': 0.4}, {
↪'n_epochs': 10, 'lr_all': 0.002, 'reg_all': 0.6}, {'n_epochs': 10,
↪'lr_all': 0.005, 'reg_all': 0.4}, {'n_epochs': 10, 'lr_all': 0.005,
↪'reg_all': 0.6}]
'param_n_epochs':    [5, 5, 5, 5, 10, 10, 10, 10]
'param_lr_all':      [0.0, 0.0, 0.01, 0.01, 0.0, 0.0, 0.01, 0.01]
'param_reg_all':     [0.4, 0.6, 0.4, 0.6, 0.4, 0.6, 0.4, 0.6]
```

As you can see, each list has the same size of the number of parameter combination. It corresponds to the following table:

split[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]reg_all
0.99[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	{'n_epochs': 5, 'lr_all': 0.002, 'reg_all': 0.4}	[illegible]	0.4
1.00[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	{'n_epochs': 5, 'lr_all': 0.002, 'reg_all': 0.6}	[illegible]	0.6
0.97[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	{'n_epochs': 5, 'lr_all': 0.005, 'reg_all': 0.4}	[illegible]	0.4
0.98[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	{'n_epochs': 5, 'lr_all': 0.005, 'reg_all': 0.6}	[illegible]	0.6
0.97[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	{'n_epochs': 10, 'lr_all': 0.002, 'reg_all': 0.4}	[illegible]	0.4
0.98[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	{'n_epochs': 10, 'lr_all': 0.002, 'reg_all': 0.6}	[illegible]	0.6
0.96[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	{'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.4}	[illegible]	0.4
0.97[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	{'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.6}	[illegible]	0.6

1.5 Command line usage

idly can also be used from the command line, for example:

```
idly -algo SVD -params "{'n_epochs': 5, 'verbose': True}" -load-
↪builtin ml-100k -n-folds 3
```

See detailed usage by running:

```
idly -h
```

CHAPTER 2

Using prediction algorithms

idly provides a bunch of built-in algorithms. All algorithms derive from the `AlgoBase` base class, where are implemented some key methods (e.g. `predict`, `fit` and `test`). The list and details of the available prediction algorithms can be found in the `prediction_algorithms` package documentation.

Every algorithm is part of the global idly namespace, so you only need to import their names from the idly package, for example:

```
from idly import KNNBasic
algo = KNNBasic()
```

Some of these algorithms may use *baseline estimates*, some may use a *similarity measure*. We will here review how to configure the way baselines and similarities are computed.

2.1 Baselines estimates configuration

Note: This section only applies to algorithms (or similarity measures) that try to minimize the following regularized squared error (or equivalent):

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - (\mu + b_u + b_i))^2 + \lambda \left(b_u^2 + b_i^2\right) .$$

For algorithms using baselines in another objective function (e.g. the `SVD` algorithm), the baseline configuration is done differently and is specific to each algorithm. Please refer to their own

documentation.

First of all, if you do not want to configure the way baselines are computed, you don't have to: the default parameters will do just fine. If you do want to well... This is for you.

You may want to read section 2.1 of *[Kor10]* to get a good idea of what are baseline estimates.

Baselines can be estimated in two different ways:

- Using Stochastic Gradient Descent (SGD).
- Using Alternating Least Squares (ALS).

You can configure the way baselines are computed using the `bsl_options` parameter passed at the creation of an algorithm. This parameter is a dictionary for which the key `'method'` indicates the method to use. Accepted values are `'als'` (default) and `'sgd'`. Depending on its value, other options may be set. For ALS:

- `'reg_i'`: The regularization parameter for items. Corresponding to λ_2 in *[Kor10]*. Default is `10`.
- `'reg_u'`: The regularization parameter for users. Corresponding to λ_3 in *[Kor10]*. Default is `15`.
- `'n_epochs'`: The number of iteration of the ALS procedure. Default is `10`. Note that in *[Kor10]*, what is described is a **single** iteration ALS process.

And for SGD:

- `'reg'`: The regularization parameter of the cost function that is optimized, corresponding to λ_1 and then λ_5 in *[Kor10]* Default is `0.02`.
- `'learning_rate'`: The learning rate of SGD, corresponding to γ in *[Kor10]*. Default is `0.005`.
- `'n_epochs'`: The number of iteration of the SGD procedure. Default is 20.

Note: For both procedures (ALS and SGD), user and item biases (b_u and b_i) are initialized to zero.

Usage examples:

Listing 2.1: From file `examples/baselines_conf.py`

```
print('Using ALS')
bsl_options = {'method': 'als',
               'n_epochs': 5,
               'reg_u': 12,
               'reg_i': 5
               }
algo = BaselineOnly(bsl_options=bsl_options)
```

Listing 2.2: From file `examples/baselines_conf.py`

```
print('Using SGD')
bsl_options = {'method': 'sgd',
               'learning_rate': .00005,
               }
algo = BaselineOnly(bsl_options=bsl_options)
```

Note that some similarity measures may use baselines, such as the `pearson_baseline` similarity. Configuration works just the same, whether the baselines are used in the actual prediction $\hat{r}_{ui}$ or not:

Listing 2.3: From file `examples/baselines_conf.py`

```
bsl_options = {'method': 'als',
               'n_epochs': 20,
               }
sim_options = {'name': 'pearson_baseline'}
algo = KNNBasic(bsl_options=bsl_options, sim_options=sim_options)
```

This leads us to similarity measure configuration, which we will review right now.

2.2 Similarity measure configuration

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

- `'name'`: The name of the similarity to use, as defined in the `similarities` module. Default is `'MSD'`.
- `'user_based'`: Whether similarities will be computed between users or between items. This has a **huge** impact on the performance of a prediction algorithm. Default is `True`.

- `'min_support'`: The minimum number of common items (when `'user_based'` is `'True'`) or minimum number of common users (when `'user_based'` is `'False'`) for the similarity not to be zero. Simply put, if $|I_{uv}| <$

Usage examples:

Listing 2.4: From file `examples/similarity_conf.py`

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

Listing 2.5: From file `examples/similarity_conf.py`

```
sim_options = {'name': 'pearson_baseline',
               'shrinkage': 0  # no shrinkage
               }
algo = KNNBasic(sim_options=sim_options)
```

See also:

The `similarities` module.

CHAPTER 3

How to build your own prediction algorithm

This page describes how to build a custom prediction algorithm using idly.

3.1 The basics

Want to get your hands dirty? Cool.

Creating your own prediction algorithm is pretty simple: an algorithm is nothing but a class derived from `AlgoBase` that has an `estimate` method. This is the method that is called by the `predict()` method. It takes in an **inner** user id, an **inner** item id (see *this note*), and returns the estimated rating $\hat{r}_{ui}$:

Listing 3.1: From file `examples/building_custom_algorithms/most_basic_algorithm.py`

```
from surprise import AlgoBase
from surprise import Dataset
from surprise.model_selection import cross_validate


class MyOwnAlgorithm(AlgoBase):

    def __init__(self):

        # Always call base method before doing anything.
        AlgoBase.__init__(self)
```

```
    def estimate(self, u, i):

        return 3


data = Dataset.load_builtin('ml-100k')
algo = MyOwnAlgorithm()

cross_validate(algo, data, verbose=True)
```

This algorithm is the dumbest we could have thought of: it just predicts a rating of 3, regardless of users and items.

If you want to store additional information about the prediction, you can also return a dictionary with given details:

```
def estimate(self, u, i):

    details = {'info1' : 'That was',
               'info2' : 'easy stuff :)'}
    return 3, details
```

This dictionary will be stored in the `prediction` as the `details` field and can be used for *later analysis*.

3.2 The `fit` method

Now, let's make a slightly cleverer algorithm that predicts the average of all the ratings of the trainset. As this is a constant value that does not depend on current user or item, we would rather compute it once and for all. This can be done by defining the `fit` method:

Listing 3.2: From file `examples/building_custom_algorithms/most_basic_algorithm2.py`

```
class MyOwnAlgorithm(AlgoBase):

    def __init__(self):

        # Always call base method before doing anything.
        AlgoBase.__init__(self)

    def fit(self, trainset):

        # Here again: call base method before doing anything.
```

```
        AlgoBase.fit(self, trainset)

        # Compute the average rating. We might as well use the
        # trainset.global_mean attribute ;)
        self.the_mean = np.mean([r for (_, _, r) in
                                 self.trainset.all_ratings()])

        return self

    def estimate(self, u, i):

        return self.the_mean
```

The `fit` method is called e.g. by the `cross_validate` function at each fold of a cross-validation process, (but you can also *call it yourself*). Before doing anything, you should call the base class `fit()` method.

Note that the `fit()` method returns `self`. This allows to use expression like `algo.fit(trainset).test(testset)`.

3.3 The `trainset` attribute

Once the base class `fit()` method has returned, all the info you need about the current training set (rating values, etc...) is stored in the `self.trainset` attribute. This is a `Trainset` object that has many attributes and methods of interest for prediction.

To illustrate its usage, let's make an algorithm that predicts an average between the mean of all ratings, the mean rating of the user and the mean rating for the item:

Listing 3.3: From file `examples/building_custom_algorithms/mean_rating_user_item.py`

```
    def estimate(self, u, i):

        sum_means = self.trainset.global_mean
        div = 1

        if self.trainset.knows_user(u):
            sum_means += np.mean([r for (_, r) in self.trainset.ur[u]])
            div += 1
        if self.trainset.knows_item(i):
            sum_means += np.mean([r for (_, r) in self.trainset.ir[i]])
            div += 1

        return sum_means / div
```

Note that it would have been a better idea to compute all the user means in the `fit` method, thus avoiding the same computations multiple times.

3.4 When the prediction is impossible

It's up to your algorithm to decide if it can or cannot yield a prediction. If the prediction is impossible, then you can raise the `PredictionImpossible` exception. You'll need to import it first:

```
from idly import PredictionImpossible
```

This exception will be caught by the `predict()` method, and the estimation $\hat{r}_{ui}$ will be set according to the `default_prediction()` method, which can be overridden. By default, it returns the average of all ratings in the trainset.

3.5 Using similarities and baselines

Should your algorithm use a similarity measure or baseline estimates, you'll need to accept `bsl_options` and `sim_options` as parameters to the `__init__` method, and pass them along to the Base class. See how to use these parameters in the *Using prediction algorithms* section.

Methods `compute_baselines()` and `compute_similarities()` can be called in the `fit` method (or anywhere else).

Listing 3.4: From file `examples/building_custom_algorithms/.with_baselines_or_sim.py`

```
class MyOwnAlgorithm(AlgoBase):

    def __init__(self, sim_options={}, bsl_options={}):

        AlgoBase.__init__(self, sim_options=sim_options,
                          bsl_options=bsl_options)

    def fit(self, trainset):

        AlgoBase.fit(self, trainset)

        # Compute baselines and similarities
        self.bu, self.bi = self.compute_baselines()
        self.sim = self.compute_similarities()

        return self
```

```
    def estimate(self, u, i):

        if not (self.trainset.knows_user(u) and self.trainset.knows_
↪item(i)):
            raise PredictionImpossible('User and/or item is unkown.')

        # Compute similarities between u and v, where v describes all␣
↪other
        # users that have also rated item i.
        neighbors = [(v, self.sim[u, v]) for (v, r) in self.trainset.
↪ir[i]]
        # Sort these neighbors by similarity
        neighbors = sorted(neighbors, key=lambda x: x[1], reverse=True)

        print('The 3 nearest neighbors of user', str(u), 'are:')
        for v, sim_uv in neighbors[:3]:
            print('user {0:} with sim {1:1.2f}'.format(v, sim_uv))

        # ... Aaaaand return the baseline estimate anyway ;)
```

Feel free to explore the prediction_algorithms package source to get an idea of what can be done.

CHAPTER 4

Notation standards, References

In the documentation, you will find the following notation:

- R : the set of all ratings.
- R_{train}, R_{test} and $\hat{R}$ denote the training set, the test set, and the set of predicted ratings.
- U : the set of all users. u and v denotes users.
- I : the set of all items. i and j denotes items.
- U_i : the set of all users that have rated item i.
- U_{ij} : the set of all users that have rated both items i and j.
- I_u : the set of all items rated by user u.
- I_{uv} : the set of all items rated by both users u and v.
- r_{ui} : the *true* rating of user u for item i.
- $\hat{r}_{ui}$: the *estimated* rating of user u for item i.
- b_{ui} : the baseline rating of user u for item i.
- μ : the mean of all ratings.
- μ_u : the mean of all ratings given by user u.
- μ_i : the mean of all ratings given to item i.
- σ_u : the standard deviation of all ratings given by user u.
- σ_i : the standard deviation of all ratings given to item i.

- $N_i^k(u)$: the k nearest neighbors of user u that have rated item i. This set is computed using a `similarity metric`.
- $N_u^k(i)$: the k nearest neighbors of item i that are rated by user u. This set is computed using a `similarity metric`.

References

Here are the papers used as references in the documentation. Links to pdf files where added when possible. A simple Google search should lead you easily to the missing ones :)

CHAPTER 5

FAQ

You will find here the Frequently Asked Questions, as well as some other use-case examples that are not part of the User Guide.

5.1 How to get the top-N recommendations for each user

Here is an example where we retrieve retrieve the top-10 items with highest rating prediction for each user in the MovieLens-100k dataset. We first train an SVD algorithm on the whole dataset, and then predict all the ratings for the pairs (user, item) that are not in the training set. We then retrieve the top-10 prediction for each user.

Listing 5.1: From file `examples/top_n_recommendations.py`

```
from collections import defaultdict

from surprise import SVD
from surprise import Dataset


def get_top_n(predictions, n=10):
    '''Return the top-N recommendation for each user from a set of␣
↪predictions.

    Args:
        predictions(list of Prediction objects): The list of␣
↪predictions, as
```

```
            returned by the test method of an algorithm.
        n(int): The number of recommendation to output for each user.␣
↪Default
            is 10.

    Returns:
    A dict where keys are user (raw) ids and values are lists of␣
↪tuples:
        [(raw item id, rating estimation), ...] of size n.
    '''

    # First map the predictions to each user.
    top_n = defaultdict(list)
    for uid, iid, true_r, est, _ in predictions:
        top_n[uid].append((iid, est))

    # Then sort the predictions for each user and retrieve the k␣
↪highest ones.
    for uid, user_ratings in top_n.items():
        user_ratings.sort(key=lambda x: x[1], reverse=True)
        top_n[uid] = user_ratings[:n]

    return top_n


# First train an SVD algorithm on the movielens dataset.
data = Dataset.load_builtin('ml-100k')
trainset = data.build_full_trainset()
algo = SVD()
algo.fit(trainset)

# Than predict ratings for all pairs (u, i) that are NOT in the␣
↪training set.
testset = trainset.build_anti_testset()
predictions = algo.test(testset)

top_n = get_top_n(predictions, n=10)

# Print the recommended items for each user
for uid, user_ratings in top_n.items():
    print(uid, [iid for (iid, _) in user_ratings])
```

5.2 How to compute precision@k and recall@k

Here is an example where we compute Precision@k and Recall@k for each user:

$$\text{Precision@k} = \frac{|\{\text{Recommended items that are relevant}\}|}{|\{\text{Recommended items}\}|} \quad \text{Recall@k} = \frac{|\{\text{Recommended items that are relevant}\}|}{|\{\text{Relevant items}\}|}$$

An item is considered relevant if its true rating r_{ui} is greater than a given threshold. An item is considered recommended if its estimated rating $\hat{r}_{ui}$ is greater than the threshold, and if it is among the k highest estimated ratings.

Listing 5.2: From file `examples/precision_recall_at_k.py`

```
from collections import defaultdict

from surprise import Dataset
from surprise import SVD
from surprise.model_selection import KFold


def precision_recall_at_k(predictions, k=10, threshold=3.5):
    '''Return precision and recall at k metrics for each user.'''

    # First map the predictions to each user.
    user_est_true = defaultdict(list)
    for uid, _, true_r, est, _ in predictions:
        user_est_true[uid].append((est, true_r))

    precisions = dict()
    recalls = dict()
    for uid, user_ratings in user_est_true.items():

        # Sort user ratings by estimated value
        user_ratings.sort(key=lambda x: x[0], reverse=True)

        # Number of relevant items
        n_rel = sum((true_r >= threshold) for (_, true_r) in user_
↪ratings)

        # Number of recommended items in top k
        n_rec_k = sum((est >= threshold) for (est, _) in user_
↪ratings[:k])

        # Number of relevant and recommended items in top k
        n_rel_and_rec_k = sum(((true_r >= threshold) and (est >=␣
↪threshold))
                              for (est, true_r) in user_ratings[:k])

        # Precision@K: Proportion of recommended items that are␣
↪relevant
```

```
        precisions[uid] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0␣
↪else 1

        # Recall@K: Proportion of relevant items that are recommended
        recalls[uid] = n_rel_and_rec_k / n_rel if n_rel != 0 else 1

    return precisions, recalls


data = Dataset.load_builtin('ml-100k')
kf = KFold(n_splits=5)
algo = SVD()

for trainset, testset in kf.split(data):
    algo.fit(trainset)
    predictions = algo.test(testset)
    precisions, recalls = precision_recall_at_k(predictions, k=5,␣
↪threshold=4)

    # Precision and recall can then be averaged over all users
    print(sum(prec for prec in precisions.values()) / len(precisions))
    print(sum(rec for rec in recalls.values()) / len(recalls))
```

5.3 How to get the k nearest neighbors of a user (or item)

You can use the `get_neighbors()` methods of the algorithm object. This is only relevant for algorithms that use a similarity measure, such as the *k-NN algorithms*.

Here is an example where we retrieve the 10 nearest neighbors of the movie Toy Story from the MovieLens-100k dataset. The output is:

```
The 10 nearest neighbors of Toy Story are:
Beauty and the Beast (1991)
Raiders of the Lost Ark (1981)
That Thing You Do! (1996)
Lion King, The (1994)
Craft, The (1996)
Liar Liar (1997)
Aladdin (1992)
Cool Hand Luke (1967)
Winnie the Pooh and the Blustery Day (1968)
Indiana Jones and the Last Crusade (1989)
```

There's a lot of boilerplate because of the conversions between movie names and their raw/inner ids (see *this note*), but it all boils down to the use of `get_neighbors()`:

Listing 5.3: From file `examples/k_nearest_neighbors.py`

```
import io  # needed because of weird encoding of u.item file

from surprise import KNNBaseline
from surprise import Dataset
from surprise import get_dataset_dir


def read_item_names():
    """Read the u.item file from MovieLens 100-k dataset and return two
    mappings to convert raw ids into movie names and movie names into␣
↪raw ids.
    """

    file_name = get_dataset_dir() + '/ml-100k/ml-100k/u.item'
    rid_to_name = {}
    name_to_rid = {}
    with io.open(file_name, 'r', encoding='ISO-8859-1') as f:
        for line in f:
            line = line.split('|')
            rid_to_name[line[0]] = line[1]
            name_to_rid[line[1]] = line[0]

    return rid_to_name, name_to_rid


# First, train the algortihm to compute the similarities between items
data = Dataset.load_builtin('ml-100k')
trainset = data.build_full_trainset()
sim_options = {'name': 'pearson_baseline', 'user_based': False}
algo = KNNBaseline(sim_options=sim_options)
algo.fit(trainset)

# Read the mappings raw id <-> movie name
rid_to_name, name_to_rid = read_item_names()

# Retrieve inner id of the movie Toy Story
toy_story_raw_id = name_to_rid['Toy Story (1995)']
toy_story_inner_id = algo.trainset.to_inner_iid(toy_story_raw_id)

# Retrieve inner ids of the nearest neighbors of Toy Story.
toy_story_neighbors = algo.get_neighbors(toy_story_inner_id, k=10)

# Convert inner ids of the neighbors into names.
toy_story_neighbors = (algo.trainset.to_raw_iid(inner_id)
                       for inner_id in toy_story_neighbors)
```

```
toy_story_neighbors = (rid_to_name[rid]
                       for rid in toy_story_neighbors)

print()
print('The 10 nearest neighbors of Toy Story are:')
for movie in toy_story_neighbors:
    print(movie)
```

Naturally, the same can be done for users with minor modifications.

5.4 How to serialize an algorithm

Prediction algorithms can be serialized and loaded back using the `dump()` and `load()` functions. Here is a small example where the SVD algorithm is trained on a dataset and serialized. It is then reloaded and can be used again for making predictions:

Listing 5.4: From file `examples/serialize_algorithm.py`

```
import os

from surprise import SVD
from surprise import Dataset
from surprise import dump


data = Dataset.load_builtin('ml-100k')
trainset = data.build_full_trainset()

algo = SVD()
algo.fit(trainset)

# Compute predictions of the 'original' algorithm.
predictions = algo.test(trainset.build_testset())

# Dump algorithm and reload it.
file_name = os.path.expanduser('~/dump_file')
dump.dump(file_name, algo=algo)
_, loaded_algo = dump.load(file_name)

# We now ensure that the algo is still the same by checking the␣
↪predictions.
predictions_loaded_algo = loaded_algo.test(trainset.build_testset())
assert predictions == predictions_loaded_algo
print('Predictions are the same')
```

Algorithms can be serialized along with their predictions, so that can be further analyzed or compared with other algorithms, using pandas dataframes. Some examples are given in the two following notebooks:

- Dumping and analysis of the KNNBasic algorithm.
- Comparison of two algorithms.

5.5 How to build my own prediction algorithm

There's a whole guide *here*.

5.6 What are raw and inner ids

Users and items have a raw id and an inner id. Some methods will use/return a raw id (e.g. the `predict()` method), while some other will use/return an inner id.

Raw ids are ids as defined in a rating file or in a pandas dataframe. They can be strings or numbers. Note though that if the ratings were read from a file which is the standard scenario, they are represented as strings. **This is important to know if you're using e.g.** `predict()` **or other methods that accept raw ids as parameters.**

On trainset creation, each raw id is mapped to a unique integer called inner id, which is a lot more suitable for idly to manipulate. Conversions between raw and inner ids can be done using the `to_inner_uid()`, `to_inner_iid()`, `to_raw_uid()`, and `to_raw_iid()` methods of the `trainset`.

5.7 Can I use my own dataset with idly, and can it be a pandas dataframe

Yes, and yes. See the *user guide*.

5.8 How to tune an algorithm parameters

You can tune the parameters of an algorithm with the `GridSearchCV` class as described *here*. After the tuning, you may want to have an *unbiased estimate of your algorithm performances*.

5.9 How to get accuracy measures on the training set

You can use the `build_testset()` method of the `Trainset` object to build a testset that can be then used with the `test()` method:

Listing 5.5: From file `examples/evaluate_on_trainset.py`

```
from surprise import Dataset
from surprise import SVD
from surprise import accuracy
from surprise.model_selection import KFold


data = Dataset.load_builtin('ml-100k')

algo = SVD()

trainset = data.build_full_trainset()
algo.fit(trainset)

testset = trainset.build_testset()
predictions = algo.test(testset)
# RMSE should be low as we are biased
accuracy.rmse(predictions, verbose=True)  # ~ 0.68 (which is low)
```

Check out the example file for more usage examples.

5.10 How to save some data for unbiased accuracy estimation

If your goal is to tune the parameters of an algorithm, you may want to spare a bit of data to have an unbiased estimation of its performances. For instance you may want to split your data into two sets A and B. A is used for parameter tuning using grid search, and B is used for unbiased estimation. This can be done as follows:

Listing 5.6: From file `examples/split_data_for_unbiased_estimation.py`

```
import random

from surprise import SVD
from surprise import Dataset
from surprise import accuracy
from surprise.model_selection import GridSearchCV
```

```
# Load the full dataset.
data = Dataset.load_builtin('ml-100k')
raw_ratings = data.raw_ratings

# shuffle ratings if you want
random.shuffle(raw_ratings)

# A = 90% of the data, B = 10% of the data
threshold = int(.9 * len(raw_ratings))
A_raw_ratings = raw_ratings[:threshold]
B_raw_ratings = raw_ratings[threshold:]

data.raw_ratings = A_raw_ratings  # data is now the set A

# Select your best algo with grid search.
print('Grid Search...')
param_grid = {'n_epochs': [5, 10], 'lr_all': [0.002, 0.005]}
grid_search = GridSearchCV(SVD, param_grid, measures=['rmse'], cv=3)
grid_search.fit(data)

algo = grid_search.best_estimator['rmse']

# retrain on the whole set A
trainset = data.build_full_trainset()
algo.fit(trainset)

# Compute biased accuracy on A
predictions = algo.test(trainset.build_testset())
print('Biased accuracy on A,', end='   ')
accuracy.rmse(predictions)

# Compute unbiased accuracy on B
testset = data.construct_testset(B_raw_ratings)  # testset is now the␣
↪set B
predictions = algo.test(testset)
print('Unbiased accuracy on B,', end=' ')
accuracy.rmse(predictions)
```

5.11 How to have reproducible experiments

Some algorithms randomly initialize their parameters (sometimes with `numpy`), and the cross-validation folds are also randomly generated. If you need to reproduce your experiments multiple times, you just have to set the seed of the RNG at the beginning of your program:

```
import random
import numpy as np

my_seed = 0
random.seed(my_seed)
numpy.random.seed(my_seed)
```

5.12 Where are datasets stored and how to change it?

By default, datasets downloaded by idly will be saved in the `'~/.idly_data'` directory. This is also where dump files will be stored. You can change the default directory by setting the `'IDLY_DATA_FOLDER'` environment variable.

CHAPTER 6

prediction_algorithms package

You may want to check the *notation standards* before diving into the formulas.

6.1 The algorithm base class

6.2 The predictions module

6.3 Basic algorithms

These are basic algorithms that do not do much work but that are still useful for comparing accuracies.

6.4 k-NN inspired algorithms

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

Note: For each of these algorithms, the actual number of neighbors that are aggregated to compute an estimation is necessarily less than or equal to k. First, there might just not exist enough neighbors and second, the sets $N_i^k(u)$ and $N_u^k(i)$ only include neighbors for which the similarity measure is **positive**. It would make no sense to aggregate ratings from users (or items) that are

negatively correlated. For a given prediction, the actual number of neighbors can be retrieved in the `'actual_k'` field of the `details` dictionary of the `prediction`.

You may want to read the *User Guide* on how to configure the `sim_options` parameter.

6.5 Matrix Factorization-based algorithms

6.6 Slope One

6.7 Co-clustering

CHAPTER 7

The model_selection package

idly provides various tools to run cross-validation procedures and search the best parameters for a prediction algorithm. The tools presented here are all heavily inspired from the excellent scikit learn library.

7.1 Cross validation iterators

7.2 Cross validation

7.3 Parameter search

CHAPTER 8

similarities module

CHAPTER 9

accuracy module

CHAPTER 10

dataset module

CHAPTER 11

Trainset class

CHAPTER 12

Reader class

CHAPTER 13

evaluate module

CHAPTER 14

dump module

Bibliography

[GM05] Thomas George and Srujana Merugu. A scalable collaborative filtering framework based on co-clustering. 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.6458&rep=rep1&type=pdf.

[Kor08] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. 2008. URL: http://www.cs.rochester.edu/twiki/pub/Main/HarpSeminar/Factorization_Meets_the_Neighborhood-_a_Multifaceted_Collaborative_Filtering_Model.pdf.

[Kor10] Yehuda Koren. Factor in the neighbors: scalable and accurate collaborative filtering. 2010. URL: http://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/a1-koren.pdf.

[KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. 2009.

[LS01] Daniel D. Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. 2001. URL: http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf.

[LM07] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering. 2007. URL: http://arxiv.org/abs/cs/0702144.

[LZXZ14] Xin Luo, Mengchu Zhou, Yunni Xia, and Qinsheng Zhu. An efficient non-negative matrix factorization-based approach to collaborative filtering for recommender systems. 2014.

[RRSK10] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. *Recommender Systems Handbook*. 1st edition, 2010.

[SM08] Ruslan Salakhutdinov and Andriy Mnih. Probabilistic matrix factorization. 2008. URL: http://papers.nips.cc/paper/3208-probabilistic-matrix-factorization.pdf.

[ZWFM96] Sheng Zhang, Weihong Wang, James Ford, and Fillia Makedon. Learning from incomplete ratings using non-negative matrix factorization. 1996. URL: http://www.siam.org/meetings/sdm06/proceedings/059zhangs2.pdf.