Orange3-Recommendation Documentation

Release 1.0.0

Salva Carrión

Sep 27, 2017

User Guide

Orange3-Recommendation is a Python library that extends Orange3 to include support for recommender systems. The code is open source, and available on github.

CHAPTER 1

Installation

Orange3-Recommendation has a couple of prerequisites that need to be installed first, but once met, the rest of picky requirements are automatically handle by the installer.

Prerequisites

Python3 + pip

Orange3-Recommendation currently requires **Python3** to run. *(Note: The algorithms have been design using Numpy, Scipy and Scikit-learn. Therefore, the algorithms could work with Python 2.7. But due to dependencies related with Orange3 and its data.Tables, Python3 must be used)*

Numpy, Scikit-learn and Orange3

The required dependencies to build the software are *Numpy >= 1.9.0*, *Scikit-Learn >= 0.16* and *Orange3*.

This is automatically handled by the installer. So you don't need to install anything else.

Install

This package uses distutils, which is the default way of installing python modules. To install in your home directory, use:

python setup.py install –user

To install for all users on Unix/Linux:

python setup.py build sudo python setup.py install

For development mode use:

python setup.py develop

Widget usage

After the installation, the widgets from this add-on are registered with Orange. To run Orange from the terminal use:

```
python3 -m Orange.canvas
```

new widgets are in the toolbox bar under *Recommendation* section.

CHAPTER 2

Tutorial

For a more visual tutorial go to Biolab's blog

Input data

This section describes how to load the data in Orange3-Recommendation.

Data format

Orange can read files in native tab-delimited format, or can load data from any of the major standard spreadsheet file type, like CSV and Excel. Native format starts with a header row with feature (column) names. Second header row gives the attribute type, which can be continuous, discrete, string or time. The third header line contains meta information to identify dependent features (class), irrelevant features (ignore) or meta features (meta). Here are the first few lines from a data set `ratings.tab`:

```
tid      user        movie       score
string   discrete    discrete    continuous
meta     row=1       col=1       class
1        Breza       HarrySally  2
2        Dana        Cvetje      5
3        Cene        Prometheus  5
4        Ksenija     HarrySally  4
5        Albert      Matrix      4
...
```

The third row is mandatory in this kind of datasets, in order to know which attributes correspond to the users (row=1) and which ones to the items (col=1). For the case of big datasets, users and items must be specified as a continuous attributes due to efficiency issues. Here are the first few lines from a data set `MovieLens100K.tab`:

```
user            movie         score         tid
continuous      continuous    continuous    time
row=1           col=1         class         meta
```

```
196             242             3               881250949
186             302             3               891717742
22              377             1               878887116
244             51              2               880606923
166             346             1               886397596
298             474             4               884182806
...
```

Loading data

Datasets can be loaded as follow:

```
import Orange
data = Orange.data.Table("ratings.tab")
```

In the add-on, several toy datasets are included: *ratings.tab, movielens100k.tab, binary_data.tab, epinions_train.tab, epinions_test.tab,...* and a few more.

Getting started

Rating pairs (user, item)

Let's presume that we want to load a dataset, train it and predict its first three pairs of (id_user, id_item)

```
import Orange
from orangecontrib.recommendation import BRISMFLearner

# Load data and train the model
data = Orange.data.Table('movielens100k.tab')
learner = BRISMFLearner(num_factors=15, num_iter=25, learning_rate=0.07, lmbda=0.1)
recommender = learner(data)

# Make predictions
prediction = recommender(data[:3])
print(prediction)
>>>
[ 3.79505151  3.75096513  1.293013 ]
```

The first three lines of code, import the Orange module, the BRISMF factorization model and loads the MovieLens100K dataset. In the next lines we instantiate the model (*learner = BRISMFLearner(...)*) and we fit the model with the loaded data.

Finally, we predict the ratings for the first three pairs (user, item) in the loaded dataset.

Recommend items for set of users

Now we want to get all the predictions (all items) for a set of users:

```
import numpy as np
indices_users = np.array([4, 12, 36])
prediction = recommender.predict_items(indices_users)
print(prediction)
```

```
>>>
[[ 1.34743879  4.61513578  3.90757263 ...,  3.03535099  4.08221699 4.26139511]
 [ 1.16652757  4.5516808   3.9867497  ...,  2.94690548  3.67274108 4.1868596 ]
 [ 2.74395768  4.04859096  4.04553826 ...,  3.22923456  3.69682699 4.95043435]]
```

This time, we've fill an array with the indices of the users to which make the predictions for all the items.

If we want as an output just the first *k* elements (do not confuse with *top best* items), we have to add the parameter *top=INTEGER* to the function

```
prediction = recommender.predict_items(indices_users, top=2)
print(prediction)
>>>
[[ 1.34743879  4.61513578]
 [ 1.16652757  4.5516808]
 [ 2.74395768  4.04859096]]
```

Evaluation

Finally, we want to known which of a list of recommender performs better on our dataset. Therefore, we perform cross-validation over a list of learners.

The first thing we need to do is to make a list of all the learners that we want to cross-validate.

```
from orangecontrib.recommendation import GlobalAvgLearner,
                                         ItemAvgLearner,
                                         UserAvgLearner,
                                         UserItemBaselineLearner
global_avg = GlobalAvgLearner()
items_avg = ItemAvgLearner()
users_avg = UserAvgLearner()
useritem_baseline = UserItemBaselineLearner()
brismf = BRISMFLearner(num_factors=15, num_iter=25, learning_rate=0.07, lmbda=0.1)
learners = [global_avg, items_avg, users_avg, useritem_baseline, brismf]
```

Once, we have the list of learners and the data loaded, we score the methods. For the case, we have scored the recommendation two measures for goodnes of fit, which they're later printed. To measure the error of the scoring, you can use all the functions defined in `Orange.evaluation`.

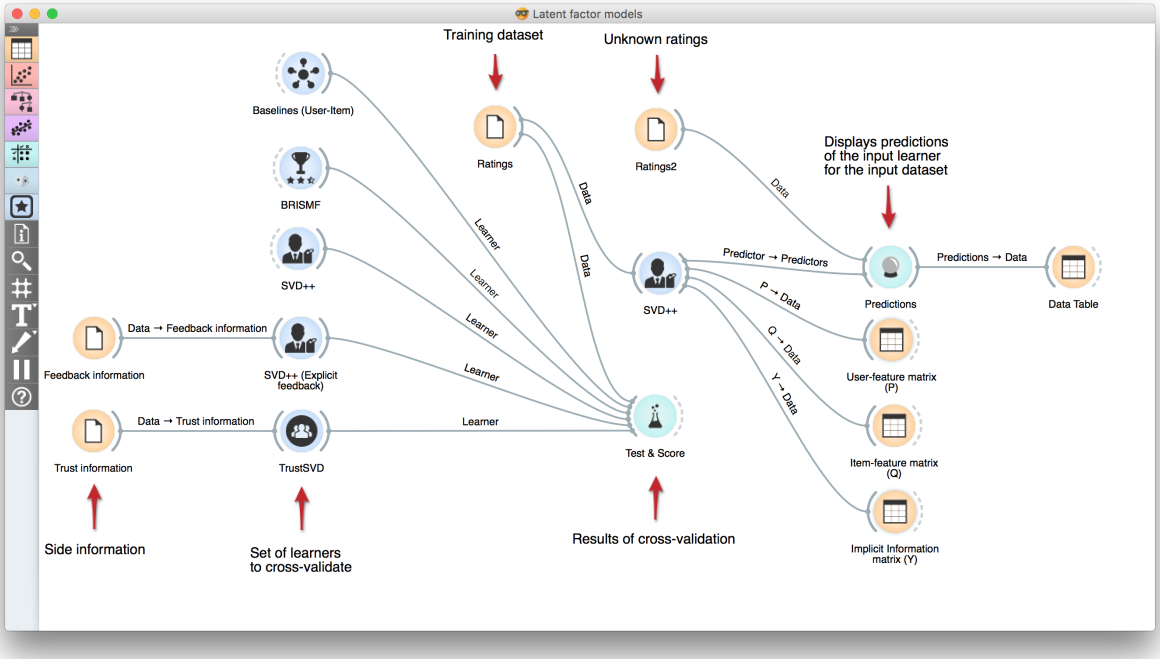
```
res = Orange.evaluation.CrossValidation(data, learners, k=5)
rmse = Orange.evaluation.RMSE(res)
r2 = Orange.evaluation.R2(res)

print("Learner  RMSE  R2")
for i in range(len(learners)):
    print("{:8s} {:.2f} {:5.2f}".format(learners[i].name, rmse[i], r2[i]))
>>>
Learner                  RMSE  R2
  - Global average       1.13 -0.00
  - Item average         1.03  0.16
  - User average         1.04  0.14
  - User-Item Baseline   0.98  0.25
  - BRISMF               0.96  0.28
```

CHAPTER 3

Frequently Asked Questions

Do I need to know to program?

Not at all. This library can be installed in Orange3 in such a way that you only need to *drag and drop* widgets to build your pipeline.



Why is there no widget for the ranking models?

Short answer: Currently Orange3 does not support ranking.

Long answer: This problem is related with how Orange3 works internally. For a given sample X, it expects to return a single value Y. The reason behind this is related with "safety", as most of the regression and classification models return just one single value.

In ranking problems, multiple results are returned. Therefore, Orange3 treats the output as the output of a classification, returning the maximun value in the sequence.

Is the library prepared for big data?

Not really. From its very beginnings we were focused on building something easy to use, mostly oriented towards educational purposes and research.

This doesn't mean that you cannot run big datasets. For instance, you can train *BRISMF* with the Netflix dataset in 30-40min. But if you plan to do so, we recommend you to use other alternatives highly optimized for those purposes.

Why are the algorithms not implemented in C/C++?

I refer back to the answer above. We want to speed-up the code as much as we can but keeping its readability and flexibility at its maximun levels, as well as having the less possible amount of dependecies.

Therefore, in order to achieve so, we try to cache as much accessings and operations as we can (keeping in mind the spacial cost), and also we try to vectorized everything we can.

Why don't you use Cython or Numba?

As it is been said before, readability and flexibility are a top priority. *Cython* is not as simple to read as *Numpy* vectorized operations and *Numba* can present problems with dependencies in some computers.

Can I contribute to the library?

Yes, please! Indeed, if you don't want, you don't have to worry neither about the widgets nor the documentation if you don't want. The only requirement is you add a new model is that it passes through all the tests.

Fork and contribute all as you want! https://github.com/biolab/orange3-recommendation

CHAPTER 4

Baselines

This widget includes four basic baseline models: Global average, User average, Item average and User-Item baseline.

Signals

Inputs:

- **Data**

  Data set.

- **Preprocessor**

  Preprocessed data.

Outputs:

- **Learner**

  The selected learner in the widget.

- **Predictor**

  Trained recommender. Signal *Predictor* sends the output signal only if input *Data* is present.

Description

- **Global average:**

  Computes the average of all ratings and use it to make predictions.

- **User average:**

  Takes the average rating value of a user to make predictions.

- **Item average:**

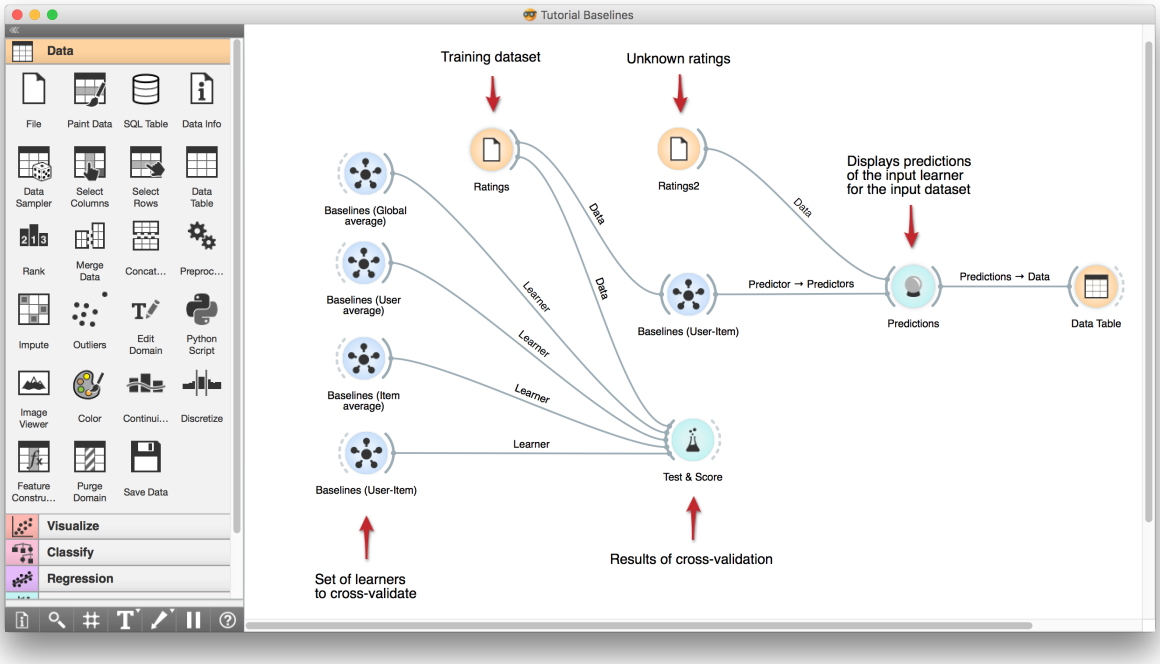
  Takes the average rating value of an item to make predictions.

- **User-Item baseline:**

  Takes the bias of users and items plus the global average to make predictions.

Example

Below is a simple workflow showing how to use both the *Predictor* and the *Learner* output. For the *Predictor* we input the prediction model into Predictions widget and view the results in Data Table. For *Learner* we can compare different learners in Test&Score widget.



CHAPTER 5

BRISMF

Matrix factorization with explicit ratings, learning is performed by stochastic gradient descent.

Signals

Inputs:

- **Data**

  Data set.

- **Preprocessor**

  Preprocessed data.

Outputs:

- **Learner**

  The learning algorithm with the supplied parameters

- **Predictor**

  Trained recommender. Signal *Predictor* sends the output signal only if input *Data* is present.

- **P**

  Latent features of the users

- **Q**

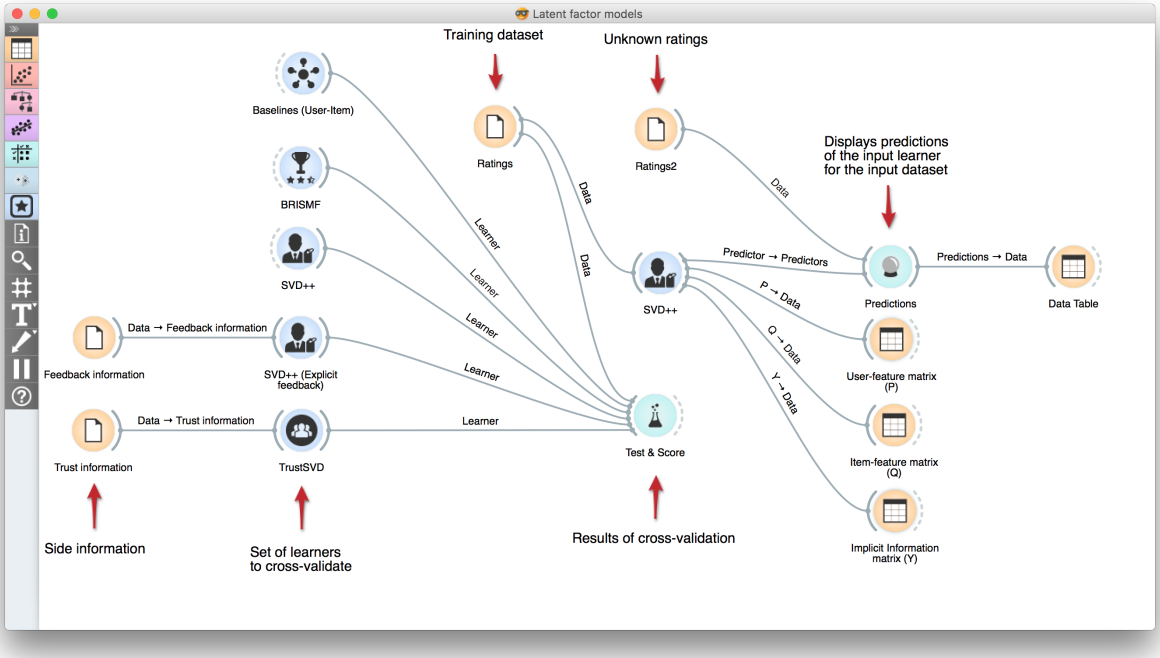
  Latent features of the items

Description

BRISMF widget uses a biased regularized algorithm to factorize a matrix into two low rank matrices as it's explained in *Y. Koren, R. Bell, C. Volinsky, Matrix Factorization Techniques for Recommender Systems. IEE Computer Society, 2009.*

Example

Below is a simple workflow showing how to use both the *Predictor* and the *Learner* output. For the *Predictor* we input the prediction model into Predictions widget and view the results in Data Table. For *Learner* we can compare different learners in Test&Score widget.



CHAPTER 6

SVD++

Matrix factorization model which makes use of implicit feedback information.

Signals

Inputs:

- **Data**

  Data set.

- **Preprocessor**

  Preprocessed data.

- **Feedback information**

  Implicit feedback information. Optional, if None (default), it will be inferred from the ratings.

Outputs:

- **Learner**

  The learning algorithm with the supplied parameters.

- **Predictor**

  Trained recommender. Signal *Predictor* sends the output signal only if input *Data* is present.

- **P**

  Latent features of the users.

- **Q**

  Latent features of the items.

- **Y**

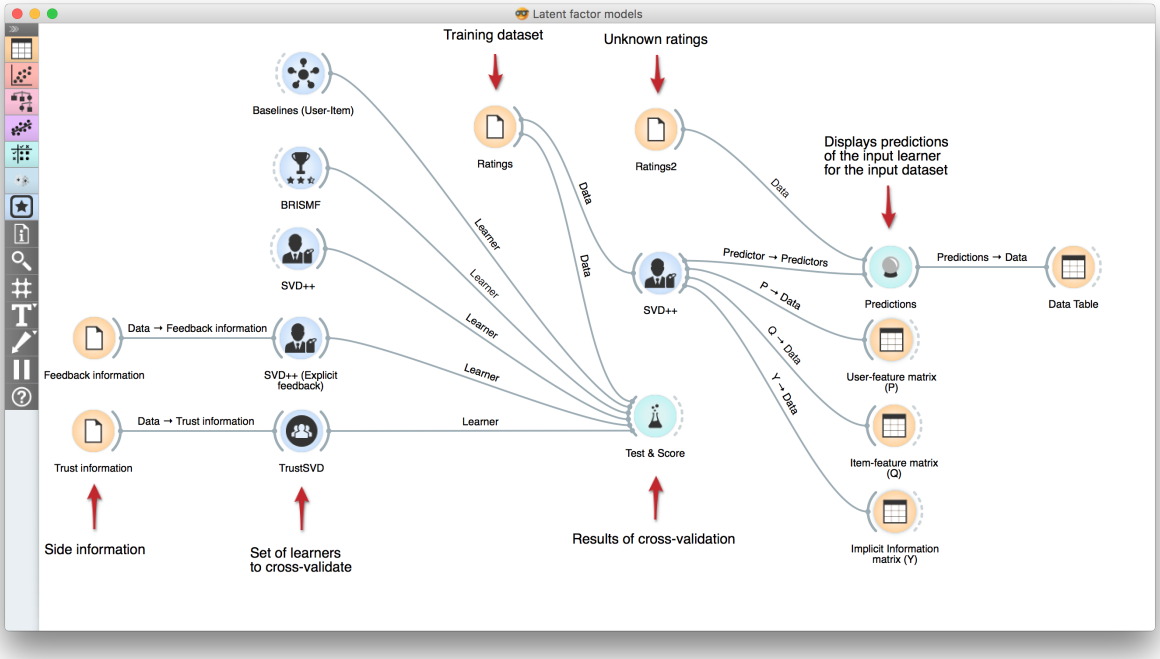
  Latent features of the implicit information.

Description

SVD++ widget uses a biased regularized algorithm which makes use of implicit feedback information to factorize a matrix into three low rank matrices as it's explained in *Y. Koren, Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model*

Example

Below is a simple workflow showing how to use both the *Predictor* and the *Learner* output. For the *Predictor* we input the prediction model into Predictions widget and view the results in Data Table. For *Learner* we can compare different learners in Test&Score widget.



CHAPTER 7

TrustSVD

Trust-based matrix factorization, which extends SVD++ with trust information.

Signals

Inputs:

- **Data**

  Data set.

- **Preprocessor**

  Preprocessed data.

- **Trust information**

  Trust information. The weights of the connections can be integer or float (binary relations can represented by 0 or 1).

Outputs:

- **Learner**

  The learning algorithm with the supplied parameters.

- **Predictor**

  Trained recommender. Signal *Predictor* sends the output signal only if input *Data* is present.

- **P**

  Latent features of the users.

- **Q**

  Latent features of the items.

- **Y**

  Latent features of the implicit information.

- **W**

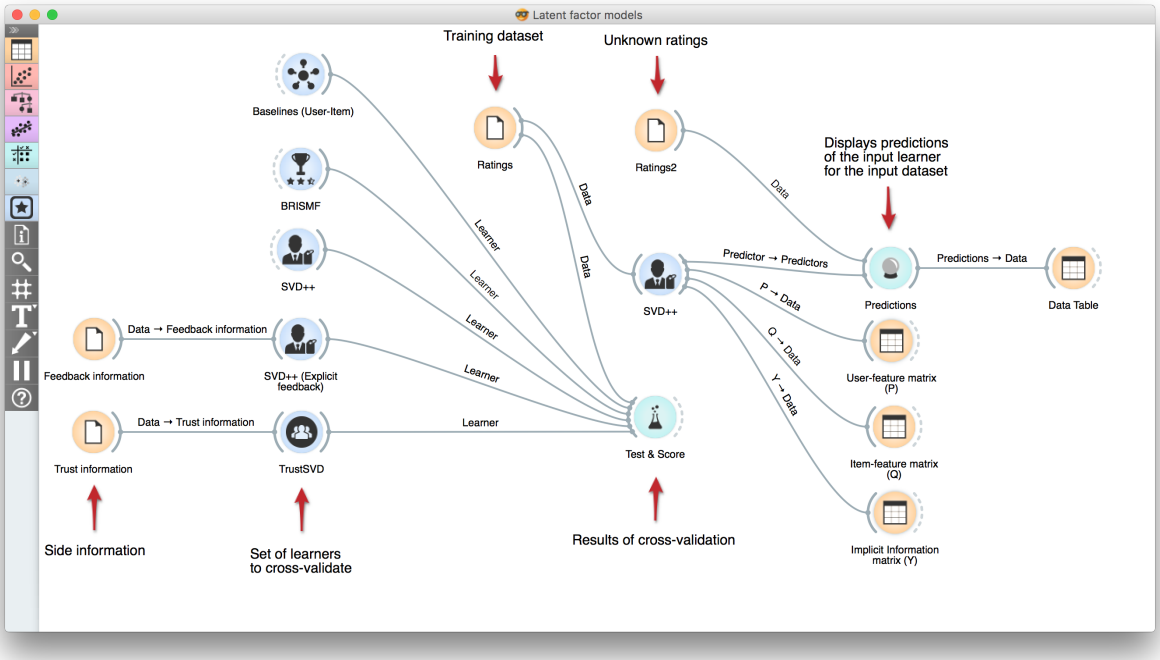
  Latent features of the trust information.

Description

TrustSVD widget uses a biased regularized algorithm which makes use of implicit feedback information and trust information to factorize a matrix into four low rank matrices as it's explained in *Guibing Guo, Jie Zhang, Neil Yorke-Smith, TrustSVD: Collaborative Filtering with Both the Explicit and Implicit Influence of User Trust and of Item Ratings*

Example

Below is a simple workflow showing how to use both the *Predictor* and the *Learner* output. For the *Predictor* we input the prediction model into Predictions widget and view the results in Data Table. For *Learner* we can compare different learners in Test&Score widget.



CHAPTER 8

Baselines (recommendation)

Global Average

Global Average uses the average rating value of all ratings to make predictions.

$$\hat{r}_{ui} = \mu$$

Example

```
import Orange
from orangecontrib.recommendation import GlobalAvgLearner

# Load data and train the model
data = Orange.data.Table('movielens100k.tab')
learner = GlobalAvgLearner()
recommender = learner(data)

prediction = recommender(data[:3])
print(prediction)
>>>
[ 3.52986  3.52986  3.52986]
```

class orangecontrib.recommendation.**GlobalAvgLearner**(*preprocessors=None*, *verbose=False*)

Global Average

This model takes the average rating value of all ratings to make predictions.

Attributes:

verbose: boolean or int, optional Prints information about the process according to the verbosity level. Values: False (verbose=0), True (verbose=1) and INTEGER

fit_storage(*data*)

Fit the model according to the given training data.

Args: data: Orange.data.Table

Returns:

self: object Returns self.

User Average

User Average uses the average rating value of a user to make predictions.

$$\hat{r}_{ui} = \mu_u$$

Example

```
import Orange
from orangecontrib.recommendation import UserAvgLearner

# Load data and train the model
data = Orange.data.Table('movielens100k.tab')
learner = UserAvgLearner()
recommender = learner(data)

# Make predictions
prediction = recommender(data[:3])
print(prediction)
>>>
[ 3.61538462  3.41304348  3.3515625 ]
```

class orangecontrib.recommendation.**UserAvgLearner**(*preprocessors=None*, *verbose=False*)
User average

This model takes the average rating value of a user to make predictions.

Attributes:

verbose: boolean or int, optional Prints information about the process according to the verbosity level. Values: False (verbose=0), True (verbose=1) and INTEGER

fit_storage(*data*)
Fit the model according to the given training data.

Args: data: Orange.data.Table

Returns:

self: object Returns self.

Item Average

Item Average uses the average rating value of an item to make predictions.

$$\hat{r}_{ui} = \mu_i$$

Example

```
import Orange
from orangecontrib.recommendation import ItemAvgLearner

# Load data and train the model
data = Orange.data.Table('movielens100k.tab')
learner = ItemAvgLearner()
recommender = learner(data)

# Make predictions
prediction = recommender(data[:3])
print(prediction)
>>>
[ 3.99145299  4.16161616  2.15384615]
```

class orangecontrib.recommendation.**ItemAvgLearner**(*preprocessors=None*, *verbose=False*)

Item average

This model takes the average rating value of an item to make predictions.

Attributes:

verbose: boolean or int, optional Prints information about the process according to the verbosity level. Values: False (verbose=0), True (verbose=1) and INTEGER

fit_storage(*data*)

Fit the model according to the given training data.

Args: data: Orange.data.Table

Returns:

self: object Returns self.

User-Item Baseline

User-Item Baseline takes the bias of users and items plus the global average to make predictions.

$$\hat{r}_{ui} = \mu + b_u + b_i$$

Example

```
import Orange
from orangecontrib.recommendation import UserItemBaselineLearner

# Load data and train the model
data = Orange.data.Table('movielens100k.tab')
learner = UserItemBaselineLearner()
recommender = learner(data)

# Make predictions
prediction = recommender(data[:3])
print(prediction)
>>>
[ 4.07697761  4.04479964  1.97554865]
```

class orangecontrib.recommendation.**UserItemBaselineLearner**(*preprocessors=None*, *verbose=False*)

User-Item baseline

This model takes the bias of users and items plus the global average to make predictions.

Attributes:

verbose: boolean or int, optional Prints information about the process according to the verbosity level. Values: False (verbose=0), True (verbose=1) and INTEGER

fit_storage(*data*)

Fit the model according to the given training data.

Args: data: Orange.data.Table

Returns:

self: object Returns self.

CHAPTER 9

Rating (`recommendation`)

BRISMF

BRISMF (Biased Regularized Incremental Simultaneous Matrix Factorization) is factorization-based algorithm for large scale recommendation systems.

The basic idea is to factorize a very sparse matrix into two low-rank matrices which represents user and item factors. This can be done by using an iterative approach to minimize the loss function.

User's predictions are defined as follows:

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

We learn the values of involved parameters by minimizing the regularized squared error function associated with:

$$\min_{p*,q*,b*} \sum_{(u,i \in k)} \left(r_{ui} - \mu - b_u - b_i - q_i^T p_u\right)^2 + \lambda({b_u}^2 + {b_i}^2 + \|p_u\|^2 + \|q_i\|^2)$$

Example

```
import Orange
from orangecontrib.recommendation import BRISMFLearner

# Load data and train the model
data = Orange.data.Table('movielens100k.tab')
learner = BRISMFLearner(num_factors=15, num_iter=25, learning_rate=0.07, lmbda=0.1)
recommender = learner(data)

# Make predictions
prediction = recommender(data[:3])
print(prediction)
>>>
[ 3.79505151  3.75096513  1.293013 ]
```

class orangecontrib.recommendation.**BRISMFLearner**(*num_factors=5*, *num_iter=25*, *learning_rate=0.07*, *bias_learning_rate=None*, *lmbda=0.1*, *bias_lmbda=None*, *min_rating=None*, *max_rating=None*, *optimizer=None*, *preprocessors=None*, *verbose=False*, *random_state=None*, *callback=None*)

BRISMF: Biased Regularized Incremental Simultaneous Matrix Factorization

This model uses stochastic gradient descent to find two low-rank matrices: user-feature matrix and item-feature matrix.

Attributes:

num_factors: int, optional The number of latent factors.

num_iter: int, optional The number of passes over the training data (aka epochs).

learning_rate: float, optional The learning rate controlling the size of update steps (general).

bias_learning_rate: float, optional The learning rate controlling the size of the bias update steps. If None (default), bias_learning_rate = learning_rate

lmbda: float, optional Controls the importance of the regularization term (general). Avoids overfitting by penalizing the magnitudes of the parameters.

bias_lmbda: float, optional Controls the importance of the bias regularization term. If None (default), bias_lmbda = lmbda

min_rating: float, optional Defines the lower bound for the predictions. If None (default), ratings won't be bounded.

max_rating: float, optional Defines the upper bound for the predictions. If None (default), ratings won't be bounded.

optimizer: Optimizer, optional Set the optimizer for SGD. If None (default), classical SGD will be applied.

verbose: boolean or int, optional Prints information about the process according to the verbosity level. Values: False (verbose=0), True (verbose=1) and INTEGER

random_state: int, optional Set the seed for the numpy random generator, so it makes the random numbers predictable. This a debbuging feature.

callback: callable Method that receives the current iteration as an argument.

fit_storage(*data*)

Fit the model according to the given training data.

Args: data: Orange.data.Table

Returns:

self: object Returns self.

SVD++

SVD++ is matrix factorization model which makes use of implicit feedback information.

User's predictions are defined as follows:

$$\hat{r}_{ui} = \mu + b_u + b_i + \left(p_u + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} y_j \right)^T q_i$$

We learn the values of involved parameters by minimizing the regularized squared error function associated with:

$$\min_{p*,q*,y*,b*} \sum_{(u,i \in k)} (r_{ui} - \mu - b_u - b_i - q_i^T \left(p_u + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} y_j \right))^2 + \lambda({b_u}^2 + {b_i}^2 + \|p_u\|^2 + \|q_i\|^2 + \sum_{j \in N(u)} \|y_j\|^2)$$

Example

```
import Orange
from orangecontrib.recommendation import SVDPlusPlusLearner

# Load data and train the model
data = Orange.data.Table('movielens100k.tab')
learner = SVDPlusPlusLearner(num_factors=15, num_iter=25, learning_rate=0.07, lmbda=0.
↪1)
recommender = learner(data)

# Make predictions
prediction = recommender(data[:3])
print(prediction)
```

class orangecontrib.recommendation.**SVDPlusPlusLearner**(*num_factors=5*, *num_iter=25*, *learning_rate=0.01*, *bias_learning_rate=None*, *lmbda=0.1*, *bias_lmbda=None*, *min_rating=None*, *max_rating=None*, *feedback=None*, *optimizer=None*, *preprocessors=None*, *verbose=False*, *random_state=None*, *callback=None*)

SVD++ matrix factorization

This model uses stochastic gradient descent to find three low-rank matrices: user-feature matrix, item-feature matrix and feedback-feature matrix.

Attributes:

num_factors: int, optional The number of latent factors.

num_iter: int, optional The number of passes over the training data (aka epochs).

learning_rate: float, optional The learning rate controlling the size of update steps (general).

bias_learning_rate: float, optional The learning rate controlling the size of the bias update steps. If None (default), bias_learning_rate = learning_rate

lmbda: float, optional Controls the importance of the regularization term (general). Avoids overfitting by penalizing the magnitudes of the parameters.

bias_lmbda: float, optional Controls the importance of the bias regularization term. If None (default), bias_lmbda = lmbda

min_rating: float, optional Defines the lower bound for the predictions. If None (default), ratings won't be bounded.

max_rating: float, optional Defines the upper bound for the predictions. If None (default), ratings won't be bounded.

feedback: Orange.data.Table Implicit feedback information. If None (default), implicit information will be inferred from the ratings (e.g.: item rated, means items seen).

optimizer: Optimizer, optional Set the optimizer for SGD. If None (default), classical SGD will be applied.

verbose: boolean or int, optional Prints information about the process according to the verbosity level. Values: False (verbose=0), True (verbose=1) and INTEGER

random_state: int, optional Set the seed for the numpy random generator, so it makes the random numbers predictable. This a debbuging feature.

callback: callable Method that receives the current iteration as an argument.

`fit_storage`(*data*)

Fit the model according to the given training data.

Args: data: Orange.data.Table

Returns:

self: object Returns self.

TrustSVD

TrustSVD is a trust-based matrix factorization, which extends SVD++ with trust information.

User's predictions are defined as follows:

$$\hat{r}_{ui} = \mu + b_u + b_i + {q_i}^\top \left(p_u + |I_u|^{-\frac{1}{2}} \sum_{i \in I_u} y_i + |T_u|^{-\frac{1}{2}} \sum_{v \in T_u} w_v \right)$$

We learn the values of involved parameters by minimizing the regularized squared error function associated with:

$$\begin{aligned}
\mathcal{L} = & \frac{1}{2} \sum_u \sum_{j \in I_u} \left(\hat{r}_{u,j} - r_{u,j}\right)^2 + \frac{\lambda_t}{2} \sum_u \sum_{v \in T_u} \left(\hat{t}_{u,v} - t_{u,v}\right)^2 \\
& + \frac{\lambda}{2} \sum_u |I_u|^{-\frac{1}{2}} b_u^2 + \frac{\lambda}{2} \sum_j |U_j|^{-\frac{1}{2}} b_j^2 \\
& + \sum_u \left(\frac{\lambda}{2} |I_u|^{-\frac{1}{2}} + \frac{\lambda_t}{2} |T_u|^{-\frac{1}{2}}\right) \|p_u\|_F^2 \\
& + \frac{\lambda}{2} \sum_j |U_j|^{-\frac{1}{2}} \|q_j\|_F^2 + \frac{\lambda}{2} \sum_i |U_i|^{-\frac{1}{2}} \|y_i\|_F^2 \\
& + \frac{\lambda}{2} \left|T_v^+\right|^{-\frac{1}{2}} \|w_v\|_F^2
\end{aligned}$$

Example

```
import Orange
from orangecontrib.recommendation import TrustSVDLearner

# Load data and train the model
ratings = Orange.data.Table('filmtrust/ratings.tab')
trust = Orange.data.Table('filmtrust/trust.tab')
learner = TrustSVDLearner(num_factors=15, num_iter=25, learning_rate=0.07,
                          lmbda=0.1, social_lmbda=0.05, trust=trust)
recommender = learner(data)

# Make predictions
prediction = recommender(data[:3])
print(prediction)
```

class orangecontrib.recommendation.**TrustSVDLearner**(*num_factors=5*, *num_iter=25*, *learning_rate=0.07*, *bias_learning_rate=None*, *lmbda=0.1*, *bias_lmbda=None*, *social_lmbda=0.05*, *min_rating=None*, *max_rating=None*, *trust=None*, *optimizer=None*, *preprocessors=None*, *verbose=False*, *random_state=None*, *callback=None*)

Trust-based matrix factorization

This model uses stochastic gradient descent to find four low-rank matrices: user-feature matrix, item-feature matrix, feedback-feature matrix and trustee-feature matrix.

Attributes:

num_factors: int, optional The number of latent factors.

num_iter: int, optional The number of passes over the training data (aka epochs).

learning_rate: float, optional The learning rate controlling the size of update steps (general).

bias_learning_rate: float, optional The learning rate controlling the size of the bias update steps. If None (default), bias_learning_rate = learning_rate

lmbda: float, optional Controls the importance of the regularization term (general). Avoids overfitting by penalizing the magnitudes of the parameters.

bias_lmbda: float, optional Controls the importance of the bias regularization term. If None (default), bias_lmbda = lmbda

social_lmbda: float, optional Controls the importance of the trust regularization term.

min_rating: float, optional Defines the lower bound for the predictions. If None (default), ratings won't be bounded.

max_rating: float, optional Defines the upper bound for the predictions. If None (default), ratings won't be bounded.

feedback: Orange.data.Table Implicit feedback information. If None (default), implicit information will be inferred from the ratings (e.g.: item rated, means items seen).

trust: Orange.data.Table Social trust information.

optimizer: Optimizer, optional Set the optimizer for SGD. If None (default), classical SGD will be applied.

verbose: boolean or int, optional Prints information about the process according to the verbosity level. Values: False (verbose=0), True (verbose=1) and INTEGER

random_state: int seed, optional Set the seed for the numpy random generator, so it makes the random numbers predictable. This a debbuging feature.

callback: callable Method that receives the current iteration as an argument.

fit_storage(*data*)

Fit the model according to the given training data.

Args: data: Orange.data.Table

Returns:

self: object Returns self.

CHAPTER 10

Ranking (`recommendation`)

CLiMF

CLiMF (Collaborative Less-is-More Filtering) is used in scenarios with binary relevance data. Hence, it's focused on improving top-k recommendations through ranking by directly maximizing the Mean Reciprocal Rank (MRR).

Following a similar technique as other iterative approaches, the two low-rank matrices can be randomly initialize and then optimize through a training loss like this:

$$F(U,V) = \sum_{i=1}^{M}\sum_{j=1}^{N} Y_{ij}[\ln \quad g(U_i^T V_i) + \sum_{k=1}^{N} \ln\left(1 - Y_{ik} g(U_i^T V_k - U_i^T V_j)\right)] - \frac{\lambda}{2}(\|U\|^2 + \|V\|^2)$$

Note: *Orange3 currently does not support ranking operations. Therefore, this model cannot be used neither in cross-validation nor in the prediction module available in Orange3*

Example

```
import Orange
import numpy as np
from orangecontrib.recommendation import CLiMFLearner

# Load data
data = Orange.data.Table('epinions_train.tab')

# Train recommender
learner = CLiMFLearner(num_factors=10, num_iter=10, learning_rate=0.0001, lmbda=0.
↪001)
recommender = learner(data)

# Load test dataset
testdata = Orange.data.Table('epinions_test.tab')

# Sample users
```

```
num_users = len(recommender.U)
num_samples = min(num_users, 1000)  # max. number to sample
users_sampled = np.random.choice(np.arange(num_users), num_samples)

# Compute Mean Reciprocal Rank (MRR)
mrr, _ = recommender.compute_mrr(data=testdata, users=users_sampled)
print('MRR: %.4f' % mrr)
>>>
MRR: 0.3975
```

class orangecontrib.recommendation.**CLiMFLearner**(*num_factors=5*, *num_iter=25*, *learning_rate=0.0001*, *lmbda=0.001*, *preprocessors=None*, *optimizer=None*, *verbose=False*, *random_state=None*, *callback=None*)

CLiMF: Collaborative Less-is-More Filtering Matrix Factorization

This model uses stochastic gradient descent to find two low-rank matrices: user-feature matrix and item-feature matrix.

CLiMF is a matrix factorization for scenarios with binary relevance data when only a few (k) items are recommended to individual users. It improves top-k recommendations through ranking by directly maximizing the Mean Reciprocal Rank (MRR).

Attributes:

num_factors: int, optional The number of latent factors.

num_iter: int, optional The number of passes over the training data (aka epochs).

learning_rate: float, optional The learning rate controlling the size of update steps (general).

lmbda: float, optional Controls the importance of the regularization term (general). Avoids overfitting by penalizing the magnitudes of the parameters.

optimizer: Optimizer, optional Set the optimizer for SGD. If None (default), classical SGD will be applied.

verbose: boolean or int, optional Prints information about the process according to the verbosity level. Values: False (verbose=0), True (verbose=1) and INTEGER

random_state: int, optional Set the seed for the numpy random generator, so it makes the random numbers predictable. This a debbuging feature.

callback: callable

fit_storage(*data*)

Fit the model according to the given training data.

Args: data: Orange.data.Table

Returns:

self: object Returns self.

CHAPTER 11

Optimizers (`recommendation.optimizers`)

The classes presented in this section are optimizers to modify the SGD updates during the training of a model.

The update functions control the learning rate during the SGD optimization

SGD	Stochastic Gradient Descent (SGD) updates
Momentum	Stochastic Gradient Descent (SGD) updates with momentum
NesterovMomentum	Stochastic Gradient Descent (SGD) updates with Nesterov momentum
AdaGrad	AdaGrad updates
RMSProp	Scale learning rates by dividing with the moving average of the root mean squared (RMS) gradients.
AdaDelta	Scale learning rates by a the ratio of accumulated gradients to accumulated step sizes, see[4] and notes for further description.
Adam	Adam updates implemented as in[5].
Adamax	Adamax updates implemented as in[6].

Stochastic Gradient Descent

This is the optimizer by default in all models.

class `orangecontrib.recommendation.optimizers.`**SGD**(*learning_rate=1.0*)

Stochastic Gradient Descent (SGD) updates

Generates update expressions of the form:

- `param := param - learning_rate * gradient`

[4] Zeiler, M. D. (2012): ADADELTA: An Adaptive Learning Rate Method. arXiv Preprint arXiv:1212.5701.
[5] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.
[6] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

Args:

learning_rate: float, optional The learning rate controlling the size of update steps

update(*grads*, *params*, *indices=None*)
SGD updates

Args:

grads: array List of gradient expressions

params: array The variables to generate update expressions for

indices: array, optional Indices in params to update

Momentum

class orangecontrib.recommendation.optimizers.**Momentum**(*learning_rate=1.0*, *momentum=0.9*)
Stochastic Gradient Descent (SGD) updates with momentum

Generates update expressions of the form:

- `velocity := momentum * velocity - learning_rate * gradient`
- `param := param + velocity`

Args:

learning_rate: float The learning rate controlling the size of update steps

momentum: float, optional The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Notes: Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by *1 - momentum*.

See Also: apply_momentum: Generic function applying momentum to updates nesterov_momentum: Nesterov's variant of SGD with momentum

update(*grads*, *params*, *indices=None*)
Momentum updates

Args:

grads: array List of gradient expressions

params: array The variables to generate update expressions for

indices: array Indices in params to update

Nesterov's Accelerated Gradient

class orangecontrib.recommendation.optimizers.**NesterovMomentum**(*learning_rate=1.0*, *momentum=0.9*)
Stochastic Gradient Descent (SGD) updates with Nesterov momentum

Generates update expressions of the form:

- `param_ahead := param + momentum * velocity`

- `velocity := momentum * velocity - learning_rate * gradient_ahead`
- `param := param + velocity`

In order to express the update to look as similar to vanilla SGD, this can be written as:

- `v_prev := velocity`
- `velocity := momentum * velocity - learning_rate * gradient`
- `param := -momentum * v_prev + (1 + momentum) * velocity`

Args:

learning_rate [float] The learning rate controlling the size of update steps

momentum: float, optional The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Notes: Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by *1 - momentum*.

The classic formulation of Nesterov momentum (or Nesterov accelerated gradient) requires the gradient to be evaluated at the predicted next position in parameter space. Here, we use the formulation described at https://github.com/lisa-lab/pylearn2/pull/136#issuecomment-10381617, which allows the gradient to be evaluated at the current parameters.

See Also: apply_nesterov_momentum: Function applying momentum to updates

update(*grads*, *params*, *indices=None*)

NAG updates

Args:

grads: array List of gradient expressions

params: array The variables to generate update expressions for

indices: array, optional Indices in params to update

Returns

updates: list of float Variables updated with the gradients

AdaGradient

class orangecontrib.recommendation.optimizers.**AdaGrad**(*learning_rate=1.0*, *epsilon=1e-06*)

AdaGrad updates

Scale learning rates by dividing with the square root of accumulated squared gradients. See[1] for further description.

- `param := param - learning_rate * gradient`

Args:

learning_rate [float or symbolic scalar] The learning rate controlling the size of update steps

epsilon: float or symbolic scalar Small value added for numerical stability

[1] Duchi, J., Hazan, E., & Singer, Y. (2011): Adaptive subgradient methods for online learning and stochastic optimization. JMLR, 12:2121-2159.

Notes: Using step size eta Adagrad calculates the learning rate for feature i at time step t as:

$$\eta_{t,i} = \frac{\eta}{\sqrt{\sum_{t'}^{t} g_{t',i}^2 + \epsilon}} g_{t,i}$$

as such the learning rate is monotonically decreasing.

Epsilon is not included in the typical formula, see[2].

References:

update(*grads*, *params*, *indices=None*)
AdaGrad updates

Args:

grads: array List of gradient expressions

params: array The variables to generate update expressions for

indices: array, optional Indices in params to update

RMSProp

class orangecontrib.recommendation.optimizers.**RMSProp**(*learning_rate=1.0*, *rho=0.9*, *epsilon=1e-06*)
Scale learning rates by dividing with the moving average of the root mean squared (RMS) gradients. See[3] for further description.

Args:

learning_rate: float The learning rate controlling the size of update steps

rho: float Gradient moving average decay factor

epsilon: float Small value added for numerical stability

Notes: *rho* should be between 0 and 1. A value of *rho* close to 1 will decay the moving average slowly and a value close to 0 will decay the moving average fast.

Using the step size η and a decay factor ρ the learning rate η_t is calculated as:

$$r_t = \rho r_{t-1} + (1 - \rho) * g^2$$
$$\eta_t = \frac{\eta}{\sqrt{r_t + \epsilon}}$$

References:

update(*grads*, *params*, *indices=None*)
RMSProp updates

Args:

grads: array List of gradient expressions

params: array The variables to generate update expressions for

indices: array, optional Indices in params to update

[2] Chris Dyer: Notes on AdaGrad. http://www.ark.cs.cmu.edu/cdyer/adagrad.pdf

[3] Tieleman, T. and Hinton, G. (2012): Neural Networks for Machine Learning, Lecture 6.5 - rmsprop. Coursera. http://www.youtube.com/watch?v=O3sxAc4hxZU (formula @5:20)

AdaDelta

class orangecontrib.recommendation.optimizers.**AdaDelta**(*learning_rate=1.0*, *rho=0.95*, *epsilon=1e-06*)

Scale learning rates by a the ratio of accumulated gradients to accumulated step sizes, see[4] and notes for further description.

Args:

learning_rate: float The learning rate controlling the size of update steps

rho: float Squared gradient moving average decay factor

epsilon: float Small value added for numerical stability

Notes: rho should be between 0 and 1. A value of rho close to 1 will decay the moving average slowly and a value close to 0 will decay the moving average fast.

rho = 0.95 and epsilon=1e-6 are suggested in the paper and reported to work for multiple datasets (MNIST, speech).

In the paper, no learning rate is considered (so learning_rate=1.0). Probably best to keep it at this value. epsilon is important for the very first update (so the numerator does not become 0).

Using the step size eta and a decay factor rho the learning rate is calculated as:

$$r_t = \rho r_{t-1} + (1-\rho) * g^2$$
$$\eta_t = \eta \frac{\sqrt{s_{t-1} + \epsilon}}{\sqrt{r_t + \epsilon}}$$
$$s_t = \rho s_{t-1} + (1-\rho) * g^2$$

References:

update(*grads*, *params*, *indices=None*)

AdaDelta updates

Args:

grads: array List of gradient expressions

params: array The variables to generate update expressions for

indices: array, optional Indices in params to update

Adam

class orangecontrib.recommendation.optimizers.**Adam**(*learning_rate=0.001*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*)

Adam updates implemented as in[5].

Args:

learning_rate [float] The learning rate controlling the size of update steps

beta_1 [float] Exponential decay rate for the first moment estimates.

beta_2 [float] Exponential decay rate for the second moment estimates.

epsilon [float] Constant for numerical stability.

Notes: The paper[5] includes an additional hyperparameter lambda. This is only needed to prove convergence of the algorithm and has no practical use, it is therefore omitted here.

References:

update(*grads*, *params*, *indices=None*)
Adam updates

Args:

grads: array List of gradient expressions

params: array The variables to generate update expressions for

indices: array, optional Indices of parameters ('params') to update. If None (default), all parameters will be updated.

Returns

updates: list of float Variables updated with the gradients

Adamax

class orangecontrib.recommendation.optimizers.**Adamax**(*learning_rate=0.001*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*)
Adamax updates implemented as in[6]. This is a variant of of the Adam algorithm based on the infinity norm.

Args:

learning_rate [float] The learning rate controlling the size of update steps

beta_1 [float] Exponential decay rate for the first moment estimates.

beta_2 [float] Exponential decay rate for the second moment estimates.

epsilon [float] Constant for numerical stability.

References:

update(*grads*, *params*, *indices=None*)
Adamax updates

Args:

grads: array List of gradient expressions

params: array The variables to generate update expressions for

indices: array, optional Indices of parameters ('params') to update. If None (default), all parameters will be updated.

Returns

updates: list of float Variables updated with the gradients

CHAPTER 12

Benchmarks

Rating

All this results refer to the training phase.

FilmTrust

Additional information:

- Loading time: 0.748s
- Training dataset: users=1,508; items=2,071; ratings=35,497; sparsity: 1.14%
- No optimizers for SGD used.

Algorithm	RMSE	MAE	Train time	Settings
Global Average	0.919	0.715	0.000s	-
Item Average	0.861	0.674	0.000s	-
User Average	0.785	0.606	0.000s	-
User-Item Baseline	0.738	0.566	0.001s	-
BRISMF	0.712	0.551	0.820s/iter	num_factors=10; num_iter=15, learning_rate=0.01; lmbda=0.1
SVD++	0.707	0.546	1.974s/iter	num_factors=10; num_iter=15; learning_rate=0.01; lmbda=0.1;
TrustSVD	0.677	0.520	3.604s/iter	num_factors=10; num_iter=15; learning_rate=0.01; lmbda=0.12; social_lmbda=0.9

MovieLens100K

Additional information:

- Loading time: 0.748s

- Training dataset: users=943; items=1,682; ratings=100,000; sparsity: 6.30%
- No optimizers for SGD used.

Algorithm	RMSE	MAE	Train time	Settings
Global Average	1.126	0.945	0.001s	-
Item Average	1.000	0.799	0.001s	-
User Average	1.031	0.826	0.001s	-
User-Item Baseline	0.938	0.738	0.001s	-
BRISMF	0.810	0.642	2.027s/it	num_factors=15; num_iter=15, learning_rate=0.07; lmbda=0.1
SVD++	0.823	0.648	7.252s/it	num_factors=15; num_iter=15; learning_rate=0.02; bias_learning_rate=0.01; lmbda=0.1; bias_lmbda=0.007

MovieLens1M

Additional information:

- Loading time: 5.144s
- Training dataset: users=6,040; items=3,706; ratings=1,000,209; sparsity: 4.47%
- No optimizers for SGD used.

Algorithm	RMSE	MAE	Train time	Settings
Global Average	1.117	0.934	0.010s	-
Item Average	0.975	0.779	0.018s	-
User Average	1.028	0.823	0.021s	-
User-Item Baseline	0.924	0.727	0.027s	-
BRISMF	0.886	0.704	19.757s/it	num_factors=15; num_iter=15, learning_rate=0.07; lmbda=0.1
SVD++	0.858	0.677	98.249s/it	num_factors=15; num_iter=15; learning_rate=0.02; bias_learning_rate=0.01; lmbda=0.1; bias_lmbda=0.007

MovieLens10M

Additional information:

- Loading time: 55.312s
- Training dataset: users=71,567; items=10,681; ratings=10,000,054; sparsity: 1.308%
- No optimizers for SGD used.

Algorithm	RMSE	MAE	Train time	Settings
Global Average	1.060	0.856	0.150s	-
Item Average	0.942	0.737	0.271s	-
User Average	0.970	0.763	0.293s	-
User-Item Baseline	0.877	0.677	0.393s	-
BRISMF	-	-	230.656s/iter	num_factors=15; num_iter=15, learning_rate=0.07; lmbda=0.1

Ranking

Epinions

Additional information:

- Loading time (training dataset): 0.094s
- Loading time (test dataset): 1.392s
- Training dataset: users=4,718; items=49,288; ratings=23,590; sparsity: 0.0101%
- Testing dataset: users=4,718; items=49,288; ratings=322,445; sparsity: 0.1386%
- No optimizers for SGD used.

Algorithm	MRR (train)	MRR (test)	Train time	Settings
CLiMF	0.0758	0.3975	1.323s/iter	num_factors=10; num_iter=10; learning_rate=0.0001; lmda=0.001

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

T

U