EvalNE Documentation

Release 0.3.4

Alexandru Mara

Jul 04, 2022

Getting started

1 Features **3**
- 1.1 For Methodologists . . . 4
- 1.2 For Practitioners . . . 4

2 Installation **7**
- 2.1 Linux/MacOS . . . 7

3 Quickstart **9**
- 3.1 As a command line tool . . . 9
- 3.2 As an API . . . 10
- 3.3 Output . . . 11
- 3.4 Parallelization . . . 12

4 API **13**
- 4.1 evalne package . . . 13

5 Release Log **79**
- 5.1 EvalNE v0.4.0 . . . 79
- 5.2 EvalNE v0.3.4 . . . 79
- 5.3 EvalNE v0.3.3 . . . 80
- 5.4 EvalNE v0.3.2 . . . 81
- 5.5 EvalNE v0.3.1 . . . 82
- 5.6 EvalNE v0.3.0 . . . 83
- 5.7 EvalNE v0.2.3 . . . 83
- 5.8 EvalNE v0.2.2 . . . 84
- 5.9 EvalNE v0.2.1 . . . 84
- 5.10 EvalNE v0.2.0 . . . 85

6 Contributing **87**

7 License **89**

8 Acknowledgements **91**

9 Help **93**

10 Citation **95**

Python Module Index **97**

Index **99**

EvalNE is an open source Python library designed for assessing and comparing the performance of Network Embedding (NE) methods on Link Prediction (LP), Sign Prediction (SP), Network Reconstruction (NR), Node Classification (NC) and vizualization downstream tasks. The library intends to simplify these complex and time consuming evaluation processes by providing automation and abstraction of tasks such as model hyper-parameter tuning and model validation, node and edge sampling, node-pair embedding computation, results reporting and data visualization, among many others. EvalNE can be used both as a command line tool and as an API and is compatible with Python 3. In its current version, EvalNE can evaluate weighted directed and undirected simple networks.

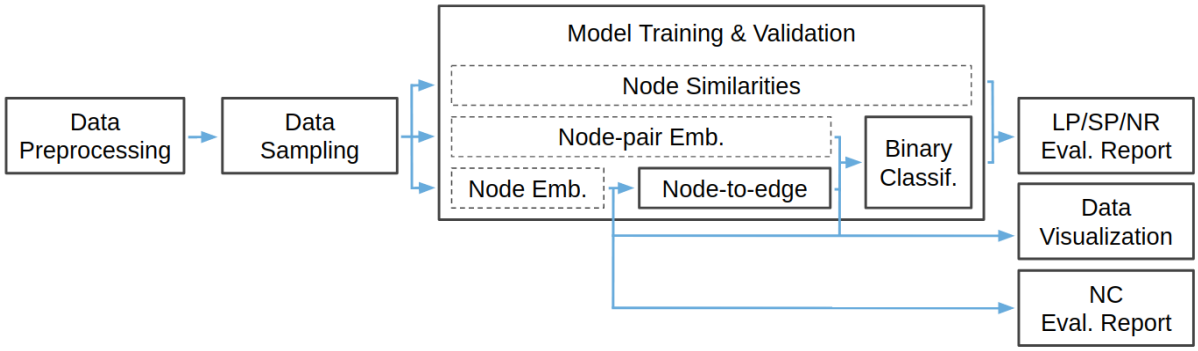
EvalNE is provided under the MIT free software licence and is maintained by Alexandru Mara (alexandru(dot)mara(at)ugent(dot)be). The source code can be found on GitHub.

See *the quickstart* to get started.

CHAPTER 1

Features

EvalNE has been designed as a pipeline of interconnected and interchangeable building blocks. This structure provides the flexibility to create different evaluation pipelines and, thus, to evaluate methods from node embeddings, node-pair embeddings or similarity scores. The main building blocks that constitute EvalNE as well as the types of tasks and methods it can evaluate are presented in the following diagram. Blocks represented with solid lines correspond to modules provided by the library and those with dashed lines are the user-specified methods to be evaluated.



Note: For node classification (NC) tasks currently only nede embedding methods are supported.

Note: The hyper-parameter tuning and evaluation setup functionalities are omitted in this diagram.

A more detailed description of the library features for the practitioner and for the methodologist are presented below. Further information can be found in our paper.

1.1 For Methodologists

A command line interface in combination with a configuration file (describing datasets, methods and evaluation setup) allows the user to evaluate any embedding method and compare it to the state of the art or replicate the experimental setup of existing papers without the need to write additional code. EvalNE does not provide implementations of any NE methods but offers the necessary environment to evaluate any off-the-shelf algorithm. Implementations of NE methods can be obtained from libraries such as OpenNE or GEM as well as directly from the web pages of the authors e.g. Deepwalk, Node2vec, LINE, PRUNE, Metapath2vec, CNE.

EvalNE also includes the following LP heuristics for both directed and undirected networks (in and out node neighbourhoods), which can be used as baselines:

- Random Prediction
- Common Neighbours
- Jaccard Coefficient
- Adamic Adar Index
- Preferential Attachment
- Resource Allocation Index
- Cosine Similarity
- Leicht-Holme-Newman index
- Topological Overlap
- Katz similarity
- All baselines (a combination of the first 5 heuristics in a 5-dim embedding)

1.2 For Practitioners

When used as an API, EvalNE provides functions to:

- Load and preprocess graphs
- Obtain general graph statistics
- Conveniently read node/edge embeddings from files
- Sample nodes/edges to form train/test/validation sets
- Different approaches for edge sampling:
  - Timestamp based sampling: latest nodes are used for testing
  - Random sampling: random split of edges in train and test sets
  - Spanning tree sampling: train set will contain a spanning tree of the graph
  - Fast depth first search sampling: similar to spanning tree but based of DFS
- Negative sampling or generation of non-edge pairs using:
  - Open world assumption: train non-edges do not overlap with train edges
  - Closed world assumption: train non-edges do not overlap with either train nor test edges
- Evaluate LP, SP and NR for methods that output:
  - Node Embeddings

  - Node-pair Embeddings
  - Similarity scores (e.g. the ones given by LP heuristics)
- Implements simple visualization routines for embeddings and graphs
- Includes NC evaluation for node embedding methods
- Provides binary operators to compute edge embeddings from node feature vectors:
  - Average
  - Hadamard
  - Weighted L1
  - Weighted L2
- Can use any scikit-learn classifier for LP/SP/NR/NC tasks
- Provides routines to run command line commands or functions with a given timeout
- Includes hyperparameter tuning based on grid search
- Implements over 10 different evaluation metrics such as AUC, F-score, etc.
- AUC and PR curves can be provided as output
- Includes routines to generate tabular outputs and directly parse them to Latex tables

CHAPTER 2

Installation

The library has been tested on Python 3.8. The supported platforms include Linux, Mac OS and Microsoft Windows.

EvalNE depends on the following open-source packages:

- Numpy
- Scipy
- Scikit-learn
- Matplotlib
- NetworkX
- Pandas
- tqdm
- kiwisolver

2.1 Linux/MacOS

Before installing EvalNE make sure that **pip** and **python-tk** packages are installed on your system, this can be done by running:

```
foo@bar:~$ sudo apt-get install python3-pip
foo@bar:~$ sudo apt-get install python3-tk
```

Option 1: Install the library using *pip*

```
foo@bar:~$ pip install evalne
```

Option 2: Cloning the code and installing

- Clone the EvalNE repository:

```
foo@bar:~$ git clone https://github.com/Dru-Mara/EvalNE.git
foo@bar:~$ cd EvalNE
```

- Install the library:

```
# System-wide install
foo@bar:~$ sudo python setup.py install

# Alterntive single user install
foo@bar:~$ python setup.py install --user
```

- Alternatively, one can first download the required dependencies and then install:

```
foo@bar:~$ pip install -r requirements.txt
foo@bar:~$ sudo python setup.py install
```

Check the installation by running *simple_example.py* or *functions_example.py* as shown below. If you have installed the package using pip, you will need to download the examples folder from the github repository first.

```
foo@bar:~$ cd examples/
foo@bar:~$ python simple_example.py
```

Note: In order to run the *evaluator_example.py* script, the OpenNE library, PRUNE and Metapath2Vec are required. Further instructions on where to obtain and how to install these methods/libraries are provided in *the quickstart* section.

CHAPTER 3

Quickstart

3.1 As a command line tool

The library takes as input an *.ini* configuration file. This file allows the user to specify the evaluation settings, from the task to perform to the networks to use, data preprocessing, methods and baselines to evaluate, and types of output to provide.

An example *conf.ini* file is provided describing the available options for each parameter. This file can be either modified to simulate different evaluation settings or used as a template to generate other *.ini* files.

Additional configuration (*.ini*) files are provided replicating the experimental sections of different papers in the NE literature. These can be found in different folders under *examples/replicated_setups*. One such configuration file is *examples/replicated_setups/node2vec/conf_node2vec.ini*. This file simulates the link prediction experiments of the paper "Scalable Feature Learning for Networks" by A. Grover and J. Leskovec.

Once the configuration is set, the evaluation can be run as indicated in the next subsection.

Running the conf examples

In order to run the evaluations using the provided *conf.ini* or any other *.ini* file, the following steps are necessary:

1. Download/Install the libraries/methods you want to test:

- For running *conf.ini*:
  - OpenNE
  - PRUNE
- For running other *.ini* files you may need:
  - Deepwalk
  - Node2vec
  - LINE
  - Metapath2Vec
  - CNE

2. Download the datasets used in the examples:

- For *conf.ini*:
  - StudentDB
  - Facebook combined network
  - Arxiv GR-QC
- For other *.ini* files you may need:
  - Facebook-wallpost
  - Arxiv Astro-Ph
  - ArXiv Hep-Ph (https://snap.stanford.edu/data/cit-HepPh.html)
  - BlogCatalog (http://socialcomputing.asu.edu/datasets/BlogCatalog3)
  - Wikipedia (http://snap.stanford.edu/node2vec)
  - PPI

3. Set the correct dataset paths in the INPATHS option of the corresponding *.ini* file. And the correct method paths under METHODS_OPNE and/or METHODS_OTHER options.

4. Run the evaluation:

```
# For conf.ini run:
foo@bar:~$ python -m evalne ./examples/conf.ini

# For conf_node2vec.ini run:
foo@bar:~$ python -m evalne ./examples/node2vec/conf_node2vec.ini
```

Note: The networks provided as input to EvalNE are required to be in edgelist format.

3.2 As an API

The library can be imported and used like any other Python module. Next, we present a very basic LP example, for more complete ones we refer the user to the *examples* folder and the docstring documentation of the evaluator and the split submodules.

```
from evalne.evaluation.evaluator import LPEvaluator
from evalne.evaluation.split import LPEvalSplit
from evalne.evaluation.score import Scoresheet
from evalne.utils import preprocess as pp

# Load and preprocess the network
G = pp.load_graph('../evalne/tests/data/network.edgelist')
G, _ = pp.prep_graph(G)

# Create an evaluator and generate train/test edge split
traintest_split = LPEvalSplit()
traintest_split.compute_splits(G)
nee = LPEvaluator(traintest_split)

# Create a Scoresheet to store the results
```

(continues on next page)

(continued from previous page)

```
scoresheet = Scoresheet()

# Set the baselines
methods = ['random_prediction', 'common_neighbours', 'jaccard_coefficient']

# Evaluate baselines
for method in methods:
    result = nee.evaluate_baseline(method=method)
    scoresheet.log_results(result)

try:
    # Check if OpenNE is installed
    import openne

    # Set embedding methods from OpenNE
    methods = ['node2vec', 'deepwalk', 'GraRep']
    commands = [
        'python -m openne --method node2vec --graph-format edgelist --p 1 --q 1',
        'python -m openne --method deepWalk --graph-format edgelist --number-walks 40
↪',
        'python -m openne --method grarep --graph-format edgelist --epochs 10']
    edge_emb = ['average', 'hadamard']

    # Evaluate embedding methods
    for i in range(len(methods)):
        command = commands[i] + " --input {} --output {} --representation-size {}"
        results = nee.evaluate_cmd(method_name=methods[i], method_type='ne',␣
↪command=command,
                                   edge_embedding_methods=edge_emb, input_delim=' ',␣
↪output_delim=' ')
        scoresheet.log_results(results)

except ImportError:
    print("The OpenNE library is not installed. Reporting results only for the␣
↪baselines...")
    pass

# Get output
scoresheet.print_tabular()
```

3.3 Output

The library stores all the output generated in a single folder per execution. The name of this folder is: *{task}_eval_{month}{day}_{hour}{min}*. Where *{task}* is one of: lp, sp, nr or nc.

The library can provide two types of outputs, depending on the value of the SCORES option of the configuration file. If the keyword *all* is specified, the library will generate a file named *eval_output.txt* containing for each method and network analysed all the metrics available (auroc, precision, f-score, etc.). If more than one experiment repeat is requested the values reported will be the average over all the repeats.

Setting the SCORES option to *%(maximize)* will generate a similar output file as before. The content of this file, however, will be a table (Alg. x Networks) containing exclusively the score specified in the MAXIMIZE option for each combination of method and network averaged over all experiment repeats. In addition a second table indicating the average execution time per method and dataset will be generated.

If the option CURVES is set to a valid option then for each method dataset and experiment repeat a PR or ROC curve will be generated. If the option SAVE_PREP_NW is set to True, each evaluated network will be stored, in edgelist format, in a folder with the same name as the network.

Finally, the library also generates an *eval.log* file and a *eval.pkl*. The first file contains important information regarding the evaluation process such as methods whose execution has failed, or validation scores. The second one encapsulates all the evaluation results as a pickle file. This file can be conveniently loaded and the results can be transformed into e.g. pandas dataframes or latex tables.

3.4 Parallelization

EvalNE makes extensive use of numpy for most operations. Numpy, in turn, uses other libraries such as OpenMP, MKL, etc., to provide parallelization. In order to allow for certain control on the maximum number of threads used during execution, we include a simple bash script (*set_numpy_threads.sh*). The script located inside the *scripts* folder can be given execution permissions and run as follows:

```
# Give execution permissions:
chmod +x set_numpy_threads.sh

# Run the script:
source set_numpy_threads.sh
# The script will then ask for the maximum number of threads to use.
```

CHAPTER 4

API

4.1 evalne package

4.1.1 Subpackages

evalne.evaluation package

Submodules

evalne.evaluation.edge_embeddings module

evalne.evaluation.edge_embeddings.**average**(*X*, *ebunch*)

Computes the embedding of each node pair (u, v) in ebunch as the element-wise average of the embeddings of nodes u and v.

Parameters

- **X** (*dict*) – A dictionary of {*nodeID*: embed_vect, *nodeID*: embed_vect, ...}. Dictionary keys are expected to be of type string and values array_like.
- **ebunch** (*iterable*) – An iterable of node pairs (u,v) for which the embeddings must be computed.

Returns **emb** – A column vector containing node-pair embeddings as rows. In the same order as ebunch.

Return type ndarray

Notes

Formally, if we use x(u) to denote the embedding corresponding to node u and x(v) to denote the embedding corresponding to node v, and if we use i to refer to the ith position in these vectors, then, the embedding of the

pair (u, v) can be computed element-wise as: $x(u,v)_i = \frac{x(u)_i + x(v)_i}{2}$. Also note that all nodeID's in ebunch must exist in X, otherwise, the method will fail.

Examples

Simple example of function use and input parameters:

```
>>> X = {'1': np.array([0, 0, 0, 0]), '2': np.array([2, 2, 2, 2]), '3': np.
↪array([1, 1, -1, -1])}
>>> ebunch = ((2, 1), (1, 1), (2, 2), (1, 3), (3, 1), (2, 3), (3, 2))
>>> average(X, ebunch)
array([[ 1. ,  1. ,  1. ,  1. ],
       [ 0. ,  0. ,  0. ,  0. ],
       [ 2. ,  2. ,  2. ,  2. ],
       [ 0.5,  0.5, -0.5, -0.5],
       [ 0.5,  0.5, -0.5, -0.5],
       [ 1.5,  1.5,  0.5,  0.5],
       [ 1.5,  1.5,  0.5,  0.5]])
```

evalne.evaluation.edge_embeddings.**compute_edge_embeddings**(*X*, *ebunch*, *method='hadamard'*)

Computes the embedding of each node pair (u, v) in ebunch as an element-wise operation on the embeddings of the end nodes u and v. The operator used is determined by the *method* parameter.

Parameters

- **X** (*dict*) – A dictionary of {*nodeID*: embed_vect, *nodeID*: embed_vect, ...}. Dictionary keys are expected to be of type string and values array_like.
- **ebunch** (*iterable*) – An iterable of node pairs (u,v) for which the embeddings must be computed.
- **method** (*string, optional*) – The operator to be used for computing the node-pair embeddings. Options are: *average*, *hadamard*, *weighted_l1* or *weighted_l2*. Default is *hadamard*.

Returns **emb** – A column vector containing node-pair embeddings as rows. In the same order as ebunch.

Return type ndarray

Examples

Simple example of function use and input parameters:

```
>>> X = {'1': np.array([0, 0, 0, 0]), '2': np.array([2, 2, 2, 2]), '3': np.
↪array([1, 1, -1, -1])}
>>> ebunch = ((2, 1), (1, 1), (2, 2), (1, 3), (3, 1), (2, 3), (3, 2))
>>> compute_edge_embeddings(X, ebunch, 'average')
array([[ 1. ,  1. ,  1. ,  1. ],
       [ 0. ,  0. ,  0. ,  0. ],
       [ 2. ,  2. ,  2. ,  2. ],
       [ 0.5,  0.5, -0.5, -0.5],
       [ 0.5,  0.5, -0.5, -0.5],
       [ 1.5,  1.5,  0.5,  0.5],
       [ 1.5,  1.5,  0.5,  0.5]])
```

evalne.evaluation.edge_embeddings.**hadamard**(*X*, *ebunch*)

Computes the embedding of each node pair (u, v) in ebunch as the element-wise product between the embeddings of nodes u and v.

Parameters

- **X** (*dict*) – A dictionary of {*nodeID*: embed_vect, *nodeID*: embed_vect, ...}. Dictionary keys are expected to be of type string and values array_like.
- **ebunch** (*iterable*) – An iterable of node pairs (u,v) for which the embeddings must be computed.

Returns **emb** – A column vector containing node-pair embeddings as rows. In the same order as ebunch.

Return type ndarray

Notes

Formally, if we use x(u) to denote the embedding corresponding to node u and x(v) to denote the embedding corresponding to node v, and if we use i to refer to the ith position in these vectors, then, the embedding of the pair (u, v) can be computed element-wise as: $x(u, v)_i = x(u)_i * x(v)_i$. Also note that all nodeID's in ebunch must exist in X, otherwise, the method will fail.

Examples

Simple example of function use and input parameters:

```
>>> X = {'1': np.array([0, 0, 0, 0]), '2': np.array([2, 2, 2, 2]), '3': np.
↪array([1, 1, -1, -1])}
>>> ebunch = ((2, 1), (1, 1), (2, 2), (1, 3), (3, 1), (2, 3), (3, 2))
>>> hadamard(X, ebunch)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 2.,  2., -2., -2.],
       [ 2.,  2., -2., -2.]])
```

evalne.evaluation.edge_embeddings.**weighted_l1**(*X*, *ebunch*)

Computes the embedding of each node pair (u, v) in ebunch as the element-wise weighted L1 distance between the embeddings of nodes u and v.

Parameters

- **X** (*dict*) – A dictionary of {*nodeID*: embed_vect, *nodeID*: embed_vect, ...}. Dictionary keys are expected to be of type string and values array_like.
- **ebunch** (*iterable*) – An iterable of node pairs (u,v) for which the embeddings must be computed.

Returns **emb** – A column vector containing node-pair embeddings as rows. In the same order as ebunch.

Return type ndarray

Notes

Formally, if we use x(u) to denote the embedding corresponding to node u and x(v) to denote the embedding corresponding to node v, and if we use i to refer to the ith position in these vectors, then, the embedding of the pair (u, v) can be computed element-wise as: $x(u, v)_i = |x(u)_i - x(v)_i|$. Also note that all nodeID's in ebunch must exist in X, otherwise, the method will fail.

Examples

Simple example of function use and input parameters:

```
>>> X = {'1': np.array([0, 0, 0, 0]), '2': np.array([2, 2, 2, 2]), '3': np.
↪array([1, 1, -1, -1])}
>>> ebunch = ((2, 1), (1, 1), (2, 2), (1, 3), (3, 1), (2, 3), (3, 2))
>>> weighted_l1(X, ebunch)
array([[2., 2., 2., 2.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 3., 3.],
       [1., 1., 3., 3.]])
```

evalne.evaluation.edge_embeddings.**weighted_l2**(*X*, *ebunch*)

Computes the embedding of each node pair (u, v) in ebunch as the element-wise weighted L2 distance between the embeddings of nodes u and v.

Parameters

- **X** (*dict*) – A dictionary of {*nodeID*: embed_vect, *nodeID*: embed_vect, ...}. Dictionary keys are expected to be of type string and values array_like.
- **ebunch** (*iterable*) – An iterable of node pairs (u,v) for which the embeddings must be computed.

Returns **emb** – A column vector containing node-pair embeddings as rows. In the same order as ebunch.

Return type ndarray

Notes

Formally, if we use x(u) to denote the embedding corresponding to node u and x(v) to denote the embedding corresponding to node v, and if we use i to refer to the ith position in these vectors, then, the embedding of the pair (u, v) can be computed element-wise as: $x(u, v)_i = (x(u)_i - x(v)_i)^2$. Also note that all nodeID's in ebunch must exist in X, otherwise, the method will fail.

Examples

Simple example of function use and input parameters:

```
>>> X = {'1': np.array([0, 0, 0, 0]), '2': np.array([2, 2, 2, 2]), '3': np.
↪array([1, 1, -1, -1])}
>>> ebunch = ((2, 1), (1, 1), (2, 2), (1, 3), (3, 1), (2, 3), (3, 2))
>>> weighted_l2(X, ebunch)
```

(continues on next page)

(continued from previous page)

```
array([[4., 4., 4., 4.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 9., 9.],
       [1., 1., 9., 9.]])
```

evalne.evaluation.evaluator module

class evalne.evaluation.evaluator.**LPEvaluator**(*traintest_split*, *trainvalid_split=None*, *dim=128*, *lp_model=LogisticRegressionCV(cv=5, scoring='roc_auc')*)

Bases: `object`

Class designed to simplify the evaluation of embedding methods for link prediction tasks.

Parameters

- **traintest_split** (LPEvalSplit) – An object containing the train graph (a subgraph of the full network that spans all nodes) and a set of train edges and non-edges. Test edges are optional. If not provided only train results will be generated.
- **trainvalid_split** (*LPEvalSplit, optional*) – An object containing the validation graph (a subgraph of the training network that spans all nodes) and a set of train and valid edges and non-edges. If not provided a split with the same parameters as the train one, but with train_frac=0.9, will be computed. Default is None.
- **dim** (*int, optional*) – Embedding dimensionality. Default is 128.
- **lp_model** (*Sklearn binary classifier, optional*) – The binary classifier to use for prediction. Default is logistic regression with 5 fold cross validation: *LogisticRegressionCV(Cs=10, cv=5, penalty='l2', scoring='roc_auc', solver='lbfgs', max_iter=100))*

Notes

In link prediction the aim is to predict, given a set of node pairs, if they should be connected or not. This is generally solved as a binary classification task. For training the binary classifier, we sample a set of edges as well as a set of unconnected node pairs. We then compute the node-pair embeddings of this training data. We use the node-pair embeddings together with the corresponding labels (0 for non-edges and 1 for edges) to train the classifier. Finally, the performance is evaluated on the test data (the remaining edges not used in training plus another set of randomly selected non-edges).

Examples

Instantiating an LPEvaluator without a specific train/validation split (this split will be computed automatically if parameter tuning for any method is required):

```
>>> from evalne.evaluation.evaluator import LPEvaluator
>>> from evalne.evaluation.split import LPEvalSplit
>>> from evalne.utils import preprocess as pp
```

(continues on next page)

(continued from previous page)

```
>>> # Load and preprocess a network
>>> G = pp.load_graph('./evalne/tests/data/network.edgelist')
>>> G, _ = pp.prep_graph(G)
>>> # Create the required train/test split
>>> traintest_split = LPEvalSplit()
>>> _ = traintest_split.compute_splits(G)
>>> # Initialize the LPEvaluator
>>> nee = LPEvaluator(traintest_split)
```

Instantiating an LPEvaluator with a specific train/validation split (allows the user to specify any parameters for the train/validation split):

```
>>> from evalne.evaluation.evaluator import LPEvaluator
>>> from evalne.evaluation.split import LPEvalSplit
>>> from evalne.utils import preprocess as pp
>>> # Load and preprocess a network
>>> G = pp.load_graph('./evalne/tests/data/network.edgelist')
>>> G, _ = pp.prep_graph(G)
>>> # Create the required train/test split
>>> traintest_split = LPEvalSplit()
>>> _ = traintest_split.compute_splits(G)
>>> # Create the train/validation split from the train data computed in the␣
↪trintest_split
>>> # The graph used to initialize this split must, thus, be the train graph from␣
↪the traintest_split
>>> trainvalid_split = EvalSplit()
>>> _ = trainvalid_split.compute_splits(traintest_split.TG)
>>> # Initialize the LPEvaluator
>>> nee = LPEvaluator(traintest_split, trainvalid_split)
```

compute_ee(*data_split*, *X*, *edge_embed_method*)

Computes node-pair embeddings using the given node embeddings dictionary and node-pair embedding method. If data_split.test_edges is None, te_edge_embeds will be None.

Parameters

- **data_split** (*a subclass of BaseEvalSplit*) – A subclass of BaseEvalSplit object that encapsulates the train/test or train/validation data.
- **X** (*dict*) – A dictionary where keys are nodes in the graph and values are the node embeddings. The keys are of type string and the values of type array.
- **edge_embed_method** (*string*) – A string indicating the method used to compute node-pair embeddings from node embeddings. The accepted values are any of the function names in evalne.evaluation.edge_embeddings.

Returns

- **tr_edge_embeds** (*matrix*) – A Numpy matrix containing the train node-pair embeddings.
- **te_edge_embeds** (*matrix*) – A Numpy matrix containing the test node-pair embeddings. Returns None if data_split.test_edges is None.

Raises AttributeError – If the node-pair embedding operator selected is not valid.

compute_pred(*data_split*, *tr_edge_embeds*, *te_edge_embeds=None*)

Computes predictions from the given node-pair embeddings. Trains an LP model with the train node-pair embeddings and performs predictions for train and test node-pair embeddings. If te_edge_embeds is None test_pred will be None.

Parameters

- **data_split** (*a subclass of BaseEvalSplit*) – A subclass of BaseEvalSplit object that encapsulates the train/test or train/validation data.
- **tr_edge_embeds** (*matrix*) – A Numpy matrix containing the train node-pair embeddings.
- **te_edge_embeds** (*matrix, optional*) – A Numpy matrix containing the test node-pair embeddings. Default is None.

Returns

- **train_pred** (*array*) – The predictions for the train data.
- **test_pred** (*array*) – The predictions for the test data. Returns None if te_edge_embeds is None.

compute_results (*data_split*, *method_name*, *train_pred*, *test_pred=None*, *label_binarizer=LogisticRegression(solver='liblinear')*, *params=None*)

Generates results from the given predictions and returns them. If test_pred is not provided, the Results object will only contain the train scores.

Parameters

- **data_split** (*a subclass of BaseEvalSplit*) – A subclass of BaseEvalSplit object that encapsulates the train/test or train/validation data.
- **method_name** (*string*) – A string indicating the name of the method for which the results will be created.
- **train_pred** – The predictions for the train data.
- **test_pred** (*array_like, optional*) – The predictions for the test data. Default is None.
- **label_binarizer** (*string or Sklearn binary classifier, optional*) – If the predictions returned by the model are not binary, this parameter indicates how these binary predictions should be computed in order to be able to provide metrics such as the confusion matrix. Any Sklear binary classifier can be used or the keyword 'median' which will used the prediction medians as binarization thresholds. Default is LogisticRegression(solver='liblinear')
- **params** (*dict, optional*) – A dictionary of parameters and values to be added to the results class. Default is None.

Returns **results** – The evaluation results.

Return type *Results*

evaluate_baseline (*method*, *neighbourhood='in'*, *timeout=None*)

Evaluates the baseline method requested. Evaluation output is returned as a Results object. For Katz neighbourhood='in' and neighbourhood='out' will return the same results corresponding to neighbourhood='in'. Execution time is contained in the results object. If the train/test split object used to initialize the evaluator does not contain test edges, the results object will only contain train results.

Parameters

- **method** (*string*) – A string indicating the name of any baseline from evalne.methods to evaluate.
- **neighbourhood** (*string, optional*) – A string indicating the 'in' or 'out' neighbourhood to be used for directed graphs. Default is 'in'.

- **timeout** (*float or None*) – A float indicating the maximum amount of time (in seconds) the evaluation can run for. If None, the evaluation is allowed to continue until completion. Default is None.

Returns **results** – The evaluation results as a Results object.

Return type *Results*

Raises

- `TimeoutExpired` – If the execution does not finish within the allocated time.
- `TypeError` – If the Katz method call is incorrect.
- `ValueError` – If the heuristic selected does not exist.

See also:

evalne.utils.util.run_function() The low level function used to run a baseline with given timeout.

Examples

Evaluating the common neighbours heuristic with default parameters. We assume an evaluator (nee) has already been instantiated (see class examples):

```
>>> result = nee.evaluate_baseline(method='common_neighbours')
>>> # Print the results
>>> result.pretty_print()
Method: common_neighbours
Parameters:
[('split_id', 0), ('dim', 128), ('eval_time', 0.06909489631652832), (
↪'neighbourhood', 'in'),
('split_alg', 'spanning_tree'), ('fe_ratio', 1.0), ('owa', True), ('nw_name',
↪'test'),
('train_frac', 0.510061919504644)]
Test scores:
tn = 1124
[...]
```

Evaluating katz with beta=0.05 and timeout 60 seconds. We assume an evaluator (nee) has already been instantiated (see class examples):

```
>>> result = nee.evaluate_baseline(method='katz 0.05', timeout=60)
>>> # Print the results
>>> result.pretty_print()
Method: katz 0.05
Parameters:
[('split_id', 0), ('dim', 128), ('eval_time', 0.11670708656311035), (
↪'neighbourhood', 'in'),
('split_alg', 'spanning_tree'), ('fe_ratio', 1.0), ('owa', True), ('nw_name',
↪'test'),
('train_frac', 0.510061919504644)]
Test scores:
tn = 1266
[...]
```

evaluate_cmd(*method_name*, *method_type*, *command*, *edge_embedding_methods*, *input_delim*, *output_delim*, *tune_params=None*, *maximize='auroc'*, *write_weights=False*, *write_dir=False*, *timeout=None*, *verbose=True*)

Evaluates an embedding method and tunes its parameters from the method's command line call string. This function can evaluate node embedding, node-pair embedding or end to end predictors.

Parameters

- **method_name** (*string*) – A string indicating the name of the method to be evaluated.
- **method_type** (*string*) – A string indicating the type of embedding method (i.e. ne, ee, e2e). NE methods are expected to return embeddings, one per graph node, as either dict or matrix sorted by nodeID. EE methods are expected to return node-pair emb. as [num_edges x embed_dim] matrix in same order as input. E2E methods are expected to return predictions as a vector in the same order as the input edgelist.
- **command** (*string*) – A string containing the call to the method as it would be written in the command line. For 'ne' methods placeholders (i.e. {}) need to be provided for the parameters: input network file, output file and embedding dimensionality, precisely IN THIS ORDER. For 'ee' methods with parameters: input network file, input train edgelist, input test edgelist, output train embeddings, output test embeddings and embedding dimensionality, 6 placeholders (i.e. {}) need to be provided, precisely IN THIS ORDER. For methods with parameters: input network file, input edgelist, output embeddings, and embedding dimensionality, 4 placeholders (i.e. {}) need to be provided, precisely IN THIS ORDER. For 'e2e' methods with parameters: input network file, input train edgelist, input test edgelist, output train predictions, output test predictions and embedding dimensionality, 6 placeholders (i.e. {}) need to be provided, precisely IN THIS ORDER. For methods with parameters: input network file, input edgelist, output predictions, and embedding dimensionality, 4 placeholders (i.e. {}) need to be provided, precisely IN THIS ORDER.
- **edge_embedding_methods** (*array-like*) – A list of methods used to compute node-pair embeddings from the node embeddings output by NE models. The accepted values are the function names in evalne.evaluation.edge_embeddings. When evaluating 'ee' or 'e2e' methods, this parameter is ignored.
- **input_delim** (*string*) – The delimiter expected by the method as input (edgelist).
- **output_delim** (*string*) – The delimiter provided by the method in the output.
- **tune_params** (*string, optional*) – A string containing all the parameters to be tuned and their values. Default is None.
- **maximize** (*string, optional*) – The score to maximize while performing parameter tuning. Default is 'auroc'.
- **write_weights** (*bool, optional*) – If True the train graph passed to the embedding methods will be stored as weighted edgelist (e.g. triplets src, dst, weight) otherwise as normal edgelist. If the graph edges have no weight attribute and this parameter is set to True, a weight of 1 will be assigned to each edge. Default is False.
- **write_dir** (*bool, optional*) – This option is only relevant for undirected graphs. If False, the train graph will be stored with a single direction of the edges. If True, both directions of edges will be stored. Default is False.
- **timeout** (*float or None, optional*) – A float indicating the maximum amount of time (in seconds) the evaluation can run for. If None, the evaluation is allowed to continue until completion. Default is None.
- **verbose** (*bool, optional*) – A parameter to control the amount of screen output. Default is True.

Returns **results** – Returns the evaluation results as a Results object.

Return type *Results*

Raises

- `TimeoutExpired` – If the execution does not finish within the allocated time.
- `IOError` – If the method call does not succeed.
- `ValueError` – If the method type is unknown. If for a method all parameter combinations fail to provide results.

See also:

evalne.utils.util.run() The low level function used to run a cmd call with given timeout.

Examples

Evaluating the OpenNE implementation of node2vec without parameter tuning and with 'average' and 'hadamard' as node-pair embedding operators. We assume the method is installed in a virtual environment and that an evaluator (nee) has already been instantiated (see class examples):

```
>>> # Prepare the cmd command for running the method. If running on a python␣
↪console full paths are required
>>> cmd = '../OpenNE-master/venv/bin/python -m openne --method node2vec '    ␣
↪   ...       '--graph-format edgelist --input {} --output {} --
↪representation-size {}'
>>> # Call the evaluation
>>> result = nee.evaluate_cmd(method_name='Node2vec', method_type='ne',␣
↪command=cmd,
...                           edge_embedding_methods=['average', 'hadamard'],␣
↪input_delim=' ', output_delim=' ')
Running command...
[...]
>>> # Print the results
>>> result.pretty_print()
Method: Node2vec
Parameters:
[('split_id', 0), ('dim', 128), ('owa', True), ('nw_name', 'test'), ('train_
↪frac', 0.51),
('split_alg', 'spanning_tree'), ('eval_time', 24.329686164855957), ('edge_
↪embed_method', 'average'),
('fe_ratio', 1.0)]
Test scores:
tn = 913
[...]
```

Evaluating the metapath2vec c++ implementation with parameter tuning and with 'average' node-pair embedding operator. We assume the method is installed and that an evaluator (nee) has already been instantiated (see class examples):

```
>>> # Prepare the cmd command for running the method. If running on a python␣
↪console full paths are required
>>> cmd = '../../methods/metapath2vec/metapath2vec -min-count 1 -iter 20 '   ␣
↪   ...       '-samples 100 -train {} -output {} -size {}'
>>> # Call the evaluation
>>> result = nee.evaluate_cmd(method_name='Metapath2vec', method_type='ne',␣
↪command=cmd,
```

(continues on next page)

(continued from previous page)

```
...                                edge_embedding_methods=['average'], input_delim=
↪' ', output_delim=' ')
Running command...
[...]
>>> # Print the results
>>> result.pretty_print()
Method: Metapath2vec
Parameters:
[('split_id', 0), ('dim', 128), ('owa', True), ('nw_name', 'test'), ('train_
↪frac', 0.51),
('split_alg', 'spanning_tree'), ('eval_time', 1.9907279014587402), ('edge_
↪embed_method', 'average'),
('fe_ratio', 1.0)]
Test scores:
tn = 919
[...]
```

evaluate_ne(*data_split*, *X*, *method*, *edge_embed_method*, *label_binarizer=LogisticRegression(solver='liblinear')*, *params=None*)

Runs the complete pipeline, from node embeddings to node-pair embeddings and returns the prediction results. If data_split.test_edges is None, the Results object will only contain train Scores.

Parameters

- **data_split** (*a subclass of BaseEvalSplit*) – A subclass of BaseEvalSplit object that encapsulates the train/test or train/validation data.
- **X** (*dict*) – A dictionary where keys are nodes in the graph and values are the node embeddings. The keys are of type string and the values of type array.
- **method** (*string*) – A string indicating the name of the method to be evaluated.
- **edge_embed_method** (*string*) – A string indicating the method used to compute node-pair embeddings from node embeddings. The accepted values are any of the function names in evalne.evaluation.edge_embeddings.
- **label_binarizer** (*string or Sklearn binary classifier, optional*) – If the predictions returned by the model are not binary, this parameter indicates how these binary predictions should be computed in order to be able to provide metrics such as the confusion matrix. Any Sklear binary classifier can be used or the keyword 'median' which will used the prediction medians as binarization thresholds. Default is *LogisticRegression(solver='liblinear')*.
- **params** (*dict, optional*) – A dictionary of parameters and values to be added to the results class. Default is None.

Returns **results** – A results object.

Return type *Results*

class evalne.evaluation.evaluator.**NCEvaluator**(*G*, *labels*, *nw_name*, *num_shuffles*, *traintest_fracs*, *trainvalid_frac*, *dim=128*, *nc_model=LogisticRegressionCV(cv=3, multi_class='ovr')*)

Bases: object

Class designed to simplify the evaluation of embedding methods for node classification tasks. The input graphs is assumed to be the entire network. Parameter tuning is performed directly on this complete graph using a train/valid node split of specified size.

Parameters

- **G** (*nx.Graph*) – The full graph for which to run the evaluation.
- **labels** (*ndarray*) – A numpy array containing nodeIDs as first columns and labels as second column.
- **nw_name** (*string*) – A string indicating the name of the network. For result logging purposes.
- **num_shuffles** (*int*) – The number of times to repeat the evaluation with different train and test node sets.
- **traintest_fracs** (*array-like*) – The fraction of all nodes to use for training.
- **trainvalid_frac** (*float*) – The fraction of all training nodes to use for actual model training (the rest are used for validation).
- **dim** (*int, optional*) – Embedding dimensionality. Default is 128.
- **nc_model** (*Sklearn binary classifier, optional*) – The classifier to use for prediction. Default is logistic regression with 3 fold cross validation: *LogisticRegressionCV(Cs=10, cv=3, penalty='l2', multi_class='ovr')*

Notes

In node multi-label classification the aim is to predict the label associated with each graph node. We start the evaluation of this task by computing the embeddings for each node in the graph. Then, we train a classifier with with a subset of these embeddings (the training nodes) and their corresponding labels. Performance is evaluate on a holdout set. For robustness, the performance is generally averaged over multiple executions over different shuffles of the data (different train and test sets). The *num_shuffles* attribute controls the number of shuffles that will be generated.

Examples

Instantiating an NCEvaluator with default parameters:

```
>>> from evalne.evaluation.evaluator import NCEvaluator
>>> from evalne.utils import preprocess as pp
>>> import numpy as np
>>> # Load and preprocess a network
>>> G = pp.load_graph('./evalne/tests/data/network.edgelist')
>>> G, _ = pp.prep_graph(G)
>>> # Generate some random node labels
>>> labels = np.random.choice([1,2,3,4,5], size=len(G.nodes))
>>> # Create pairs of (nodeID, label) and make them a column vector
>>> nl_pairs = np.vstack((range(len(G.nodes)), labels)).T
>>> # For NC we do not need to create a train test edge split, we can initialize␣
↪the evaluator directly
>>> nee = NCEvaluator(G, labels=nl_pairs, nw_name='test_network', num_shuffles=5,␣
↪traintest_fracs=[0.8, 0.5],
...                   trainvalid_frac=0.5)
```

compute_pred(*X_train*, *y_train*, *X_test=None*)

Computes predictions from the given embeddings. Trains a NC model with the train node-pair embeddings and performs predictions for train and test embeddings. If te_edge_embeds is None test_pred will be None.

Parameters

- **X_train** (*ndarray*) – An array containing the train embeddings.
- **y_train** (*ndarray*) – An array containing the train labels.
- **X_test** (*ndarray, optional*) – An array containing the test embeddings.

Returns

- **train_pred** (*ndarray*) – The label predictions for the train data.
- **test_pred** (*ndarray*) – The label predictions for the test data. Returns None if X_test is None.

compute_results(*method_name*, *train_pred*, *train_labels*, *test_pred=None*, *test_labels=None*, *params=None*)

Generates results from the given predictions and returns them. If test_pred is not provided, the Results object will only contain the train scores.

Parameters

- **method_name** (*string*) – A string indicating the name of the method for which the results will be created.
- **train_pred** (*ndarray*) – The predictions for the train data.
- **test_pred** (*ndarray, optional*) – The predictions for the test data. Default is None.
- **params** (*dict, optional*) – A dictionary of parameters and values to be added to the results class. Default is None.

Returns **results** – The evaluation results.

Return type *Results*

evaluate_cmd(*method_name*, *command*, *input_delim*, *output_delim*, *tune_params=None*, *maximize='f1_micro'*, *write_weights=False*, *write_dir=False*, *timeout=None*, *verbose=True*)

Evaluates an embedding method and tunes its parameters from the method's command line call string. Currently, this function can only evaluate node embedding methods.

Parameters

- **method_name** (*string*) – A string indicating the name of the method to be evaluated.
- **command** (*string*) – A string containing the call to the method as it would be written in the command line. For 'ne' methods placeholders (i.e. {}) need to be provided for the parameters: input network file, output file and embedding dimensionality, precisely IN THIS ORDER.
- **input_delim** (*string*) – The delimiter expected by the method as input (edgelist).
- **output_delim** (*string*) – The delimiter provided by the method in the output.
- **tune_params** (*string, optional*) – A string containing all the parameters to be tuned and their values. Default is None.
- **maximize** (*string, optional*) – The score to maximize while performing parameter tuning. Default is 'f1_micro'.
- **write_weights** (*bool, optional*) – If True the train graph passed to the embedding methods will be stored as weighted edgelist (e.g. triplets src, dst, weight) otherwise as normal edgelist. If the graph edges have no weight attribute and this parameter is set to True, a weight of 1 will be assigned to each edge. Default is False.

- **write_dir** (*bool, optional*) – This option is only relevant for undirected graphs. If False, the train graph will be stored with a single direction of the edges. If True, both directions of edges will be stored. Default is False.
- **timeout** (*float or None*) – A float indicating the maximum amount of time (in seconds) the evaluation can run for. If None, the evaluation is allowed to continue until completion. Default is None.
- **verbose** (*bool, optional*) – A parameter to control the amount of screen output. Default is True.

Returns **results** – Returns the evaluation results as a list of Results objects (one for each train-test_frac requested and each shuffle). The length of the list returned will thus be *num_shuffles * len(traintest_fracs)*.

Return type list of Results

Raises

- `TimeoutExpired` – If the execution does not finish within the allocated time.
- `IOError` – If the method call does not succeed.

See also:

evalne.utils.util.run() The low level function used to run a cmd call with given timeout.

Examples

Evaluating the OpenNE implementation of node2vec without parameter tuning. We assume the method is installed in a virtual environment and that an evaluator (nee) has already been instantiated (see class examples):

```
>>> # Prepare the cmd command for running the method. If running on a python␣
↪console full paths are required
>>> cmd = '../OpenNE-master/venv/bin/python -m openne --method node2vec '␣
↪   ...       '--graph-format edgelist --input {} --output {} --␣
↪representation-size {}'
>>> # Call the evaluation
>>> result = nee.evaluate_cmd(method_name='Node2vec', command=cmd, input_␣
↪delim=' ', output_delim=' ')
Running command...
[...]
>>> # Check the results of the first data shuffle of traintest_frac=0.8
>>> result[0].pretty_print()
Method: Node2vec_0.8
Parameters:
[('dim', 128), ('nw_name', 'test_network'), ('eval_time', 33.22737193107605)]
Test scores:
f1_micro = 0.177304964539
f1_macro = 0.0975922953451
f1_weighted = 0.107965347267
>>> # Check the results of the first data shuffle of traintest_frac=0.5
>>> result[5].pretty_print()
Method: Node2vec_0.5
Parameters:
[('dim', 128), ('nw_name', 'test_network'), ('eval_time', 33.22737193107605)]
Test scores:
f1_micro = 0.173295454545
```

(continues on next page)

(continued from previous page)

```
f1_macro = 0.0590799031477
f1_weighted = 0.0511913933524
```

Evaluating the metapath2vec c++ implementation without parameter tuning. We assume the method is installed and that an evaluator (nee) has already been instantiated (see class examples):

```
>>> # Prepare the cmd command for running the method. If running on a python␣
↪console full paths are required
>>> cmd = '../../methods/metapath2vec/metapath2vec -min-count 1 -iter 20 '    ␣
↪    ...       '-samples 100 -train {} -output {} -size {}'
>>> # Call the evaluation
>>> result = nee.evaluate_cmd(method_name='Metapath2vec', command=cmd, input_
↪delim=' ', output_delim=' ')
Running command...
[...]
>>> # Check the results of the second data shuffle of traintest_frac=0.8
>>> result.pretty_print()
Method: Metapath2vec_0.8
Parameters:
[('dim', 128), ('nw_name', 'test_network'), ('eval_time', 23.914228916168213)]
Test scores:
f1_micro = 0.205673758865
f1_macro = 0.0711656441718
f1_weighted = 0.0807553409041
>>> # Check the results of the second data shuffle of traintest_frac=0.5
>>> result.pretty_print()
Method: Metapath2vec_0.5
Parameters:
[('dim', 128), ('nw_name', 'test_network'), ('eval_time', 23.914228916168213)]
Test scores:
f1_micro = 0.215909090909
f1_macro = 0.0710280373832
f1_weighted = 0.0766779949023
```

evaluate_ne(*X*, *method_name*, *params=None*)

Runs the NC evaluation pipeline. For each 'node_frac' trains a nc_model and uses it to compute predictions which are then returned as a results object. If data_split.test_edges is None, the Results object will only contain train Scores.

Parameters

- **X** (*dict*) – A dictionary where keys are nodes in the graph and values are the node embeddings. The keys are of type string and the values of type array.
- **method_name** (*string*) – A string indicating the name of the method to be evaluated.
- **params** (*dict, optional*) – A dictionary of parameters and values to be added to the results class. Default is None.

Returns **results** – Returns a list of Results objects one per each train/test fraction and each node shuffle.

Return type list

class evalne.evaluation.evaluator.**NREvaluator**(*traintest_split*, *dim=128*, *lp_model=LogisticRegressionCV(cv=5, scoring='roc_auc')*)

Bases: *evalne.evaluation.evaluator.LPEvaluator*

Class designed to simplify the evaluation of embedding methods for network reconstruction tasks. The train graph is assumed to be the entire network. Parameter tuning is performed directly on this complete graph.

Parameters

- **traintest_split** (NREvalSplit) – An object containing the train graph (in this case the full network) and a set of train edges and non-edges. These edges can be all edges in the graph or a subset.
- **dim** (*int, optional*) – Embedding dimensionality. Default is 128.
- **lp_model** (*Sklearn binary classifier, optional*) – The binary classifier to use for prediction. Default is logistic regression with 5 fold cross validation: *LogisticRegressionCV(Cs=10, cv=5, penalty='l2', scoring='roc_auc', solver='lbfgs', max_iter=100))*

Notes

In network reconstruction the aim is to asses how well an embedding method captures the structure of a given graph. The embedding methods are trained on a complete input graph. Hyperparameter tuning is performed directly on this graph (overfitting is, in this case, expected and desired). The embeddings obtained are used to perform link predictions and their quality is evaluated. Checking the link predictions for all node pairs is generally unfeasible, therefore a subset of all node pairs in the input graph are selected for evaluation.

Examples

Instantiating an NREvaluator with default parameters (for this task train/validation splits are not necessary):

```
>>> from evalne.evaluation.evaluator import NREvaluator
>>> from evalne.evaluation.split import NREvalSplit
>>> from evalne.utils import preprocess as pp
>>> # Load and preprocess a network
>>> G = pp.load_graph('./evalne/tests/data/network.edgelist')
>>> G, _ = pp.prep_graph(G)
>>> # Create the required train/test split
>>> traintest_split = NREvalSplit()
>>> _ = traintest_split.compute_splits(G)
>>> # Initialize the NREvaluator
>>> nee = NREvaluator(traintest_split)
```

Instantiating an NREvaluator where we randomly select 10% of all node pairs in the network for evaluation:

```
>>> from evalne.evaluation.evaluator import NREvaluator
>>> from evalne.evaluation.split import NREvalSplit
>>> from evalne.utils import preprocess as pp
>>> # Load and preprocess a network
>>> G = pp.load_graph('./evalne/tests/data/network.edgelist')
>>> G, _ = pp.prep_graph(G)
>>> # Create the required train/test split and sample 0.1, i.e. 10% of all nodes
>>> traintest_split = NREvalSplit()
>>> _ = traintest_split.compute_splits(G, samp_frac=0.1)
>>> # Initialize the NREvaluator
>>> nee = NREvaluator(traintest_split)
```

evaluate_cmd(*method_name*, *method_type*, *command*, *edge_embedding_methods*, *input_delim*, *output_delim*, *tune_params=None*, *maximize='auroc'*, *write_weights=False*, *write_dir=False*, *timeout=None*, *verbose=True*)

Evaluates an embedding method and tunes its parameters from the method's command line call string. This function can evaluate node embedding, node-pair embedding or end to end predictors. If model parameter tuning is required, models are tuned directly on the train data. The returned Results object will only contain train scores.

Parameters

- **method_name** (*string*) – A string indicating the name of the method to be evaluated.
- **method_type** (*string*) – A string indicating the type of embedding method (i.e. ne, ee, e2e). NE methods are expected to return embeddings, one per graph node, as either dict or matrix sorted by nodeID. EE methods are expected to return node-pair emb. as [num_edges x embed_dim] matrix in same order as input. E2E methods are expected to return predictions as a vector in the same order as the input edgelist.
- **command** (*string*) – A string containing the call to the method as it would be written in the command line. For 'ne' methods placeholders (i.e. {}) need to be provided for the parameters: input network file, output file and embedding dimensionality, precisely IN THIS ORDER. For 'ee' methods with parameters: input network file, input train edgelist, input test edgelist, output train embeddings, output test embeddings and embedding dimensionality, 6 placeholders (i.e. {}) need to be provided, precisely IN THIS ORDER. For methods with parameters: input network file, input edgelist, output embeddings, and embedding dimensionality, 4 placeholders (i.e. {}) need to be provided, precisely IN THIS ORDER. For 'e2e' methods with parameters: input network file, input train edgelist, input test edgelist, output train predictions, output test predictions and embedding dimensionality, 6 placeholders (i.e. {}) need to be provided, precisely IN THIS ORDER. For methods with parameters: input network file, input edgelist, output predictions, and embedding dimensionality, 4 placeholders (i.e. {}) need to be provided, precisely IN THIS ORDER.
- **edge_embedding_methods** (*array-like*) – A list of methods used to compute node-pair embeddings from the node embeddings output by NE models. The accepted values are the function names in evalne.evaluation.edge_embeddings. When evaluating 'ee' or 'e2e' methods, this parameter is ignored.
- **input_delim** (*string*) – The delimiter expected by the method as input (edgelist).
- **output_delim** (*string*) – The delimiter provided by the method in the output.
- **tune_params** (*string, optional*) – A string containing all the parameters to be tuned and their values. Default is None.
- **maximize** (*string, optional*) – The score to maximize while performing parameter tuning. Default is 'auroc'.
- **write_weights** (*bool, optional*) – If True the train graph passed to the embedding methods will be stored as weighted edgelist (e.g. triplets src, dst, weight) otherwise as normal edgelist. If the graph edges have no weight attribute and this parameter is set to True, a weight of 1 will be assigned to each edge. Default is False.
- **write_dir** (*bool, optional*) – This option is only relevant for undirected graphs. If False, the train graph will be stored with a single direction of the edges. If True, both directions of edges will be stored. Default is False.
- **timeout** (*float or None, optional*) – A float indicating the maximum amount of time (in seconds) the evaluation can run for. If None, the evaluation is allowed to continue until completion. Default is None.
- **verbose** (*bool, optional*) – A parameter to control the amount of screen output. Default is True.

Returns **results** – The evaluation results as a Results object.

Return type *Results*

Raises

- `TimeoutExpired` – If the execution does not finish within the allocated time.
- `IOError` – If the method call does not succeed.
- `ValueError` – If the method type is unknown. If for a method all parameter combinations fail to provide results.

See also:

evalne.utils.util.run() The low level function used to run a cmd call with given timeout.

Examples

Evaluating the OpenNE implementation of node2vec without parameter tuning and with 'average' and 'hadamard' as node-pair embedding operators. We assume the method is installed in a virtual environment and that an evaluator (nee) has already been instantiated (see class examples):

```
>>> # Prepare the cmd command for running the method. If running on a python␣
↪console full paths are required
>>> cmd = '../OpenNE-master/venv/bin/python -m openne --method node2vec '    ␣
↪   ...       '--graph-format edgelist --input {} --output {} --
↪representation-size {}'
>>> # Call the evaluation
>>> result = nee.evaluate_cmd(method_name='Node2vec', method_type='ne',␣
↪command=cmd,
...                           edge_embedding_methods=['average', 'hadamard'],␣
↪input_delim=' ', output_delim=' ')
Running command...
[...]
>>> # Print the results
>>> result.pretty_print()
Method: Node2vec
Parameters:
[('split_id', 0), ('dim', 128), ('eval_time', 21.773473024368286), ('nw_name',
↪ 'test'),
('split_alg', 'random_edge_sample'), ('train_frac', 1), ('edge_embed_method',
↪'hadamard'), ('samp_frac', 0.01)]
Train scores:
tn = 2444
[...]
```

Evaluating the metapath2vec c++ implementation with parameter tuning and with 'average' node-pair embedding operator. We assume the method is installed and that an evaluator (nee) has already been instantiated (see class examples):

```
>>> # Prepare the cmd command for running the method. If running on a python␣
↪console full paths are required
>>> cmd = '../../methods/metapath2vec/metapath2vec -min-count 1 -iter 20 '   ␣
↪    ...       '-samples 100 -train {} -output {} -size {}'
>>> # Call the evaluation
>>> result = nee.evaluate_cmd(method_name='Metapath2vec', method_type='ne',␣
↪command=cmd,
...                           edge_embedding_methods=['average'], input_delim=
↪' ', output_delim=' ')
```

(continues on next page)

(continued from previous page)

```
Running command...
[...]
>>> # Print the results
>>> result.pretty_print()
Method: Metapath2vec
Parameters:
Method: Metapath2vec
Parameters:
[('split_id', 0), ('dim', 128), ('eval_time', 1.948814868927002), ('nw_name',
↪'test'),
('split_alg', 'random_edge_sample'), ('train_frac', 1), ('edge_embed_method',
↪'average'), ('samp_frac', 0.01)]
Train scores:
tn = 2444
[...]
```

class evalne.evaluation.evaluator.**SPEvaluator**(*traintest_split*, *trainvalid_split=None*, *dim=128*, *lp_model=LogisticRegressionCV(cv=5, scoring='roc_auc')*)

Bases: *evalne.evaluation.evaluator.LPEvaluator*

Class designed to simplify the evaluation of embedding methods for sign prediction tasks. The train and validation graphs are assumed to be weighted and contain positive and negative edges. This is a simple extension of an LP evaluation which overrides the baseline implementation to work for sign prediction.

Parameters

- **traintest_split** (SPEvalSplit) – An object containing the train graph (a subgraph of the full network that spans all nodes) and a set of train positive and negative edges. Test edges are optional. If not provided only train results will be generated.
- **trainvalid_split** (SPEvalSplit, *optional*) – An object containing the validation graph (a subgraph of the training network that spans all nodes) and a set of positive and negative edges. If not provided a split with the same parameters as the train one, but with train_frac=0.9, will be computed. Default is None.
- **dim** (*int, optional*) – Embedding dimensionality. Default is 128.
- **lp_model** (*Sklearn binary classifier, optional*) – The binary classifier to use for prediction. Default is logistic regression with 5 fold cross validation: *LogisticRegressionCV(Cs=10, cv=5, penalty='l2', scoring='roc_auc', solver='lbfgs', max_iter=100))*.

Notes

In sign prediction the aim is to predict the sign (positive or negative) of given edges. The existence of the edges is assumed (i.e. we do not predict the sign of unconnected node pairs). Therefore, sign prediction is also a binary classification task similar to link prediction where, instead of predicting the existence of edges or not, we predict the signs for edges we know exist. Unlike for link prediction, in this case we do not need to perform negative sampling, since we already have both classes (the positively and the negatively connected node pairs).

Examples

Instantiating an SPEvaluator without a specific train/validation split (this split will be computed automatically if parameter tuning for any method is required):

```
>>> from evalne.evaluation.evaluator import SPEvaluator
>>> from evalne.evaluation.split import SPEvalSplit
>>> from evalne.utils import preprocess as pp
>>> # Load and preprocess a network
>>> G = pp.load_graph('./evalne/tests/data/sig_network.edgelist')
>>> G, _ = pp.prep_graph(G)
>>> # Create the required train/test split
>>> traintest_split = SPEvalSplit()
>>> _ = traintest_split.compute_splits(G)
>>> # Check that the train test parameters are indeed the correct ones
>>> traintest_split.get_parameters()
{'split_id': 0, 'nw_name': 'test', 'split_alg': 'spanning_tree', 'train_frac': 0.
↪4980}
>>> # Initialize the SPEvaluator
>>> nee = SPEvaluator(traintest_split)
```

Instantiating an SPEvaluator with a specific train/validation split (allows the user to specify any parameters for the train/validation split). Use 'fast' as the algorithm to split train and test edges and set train fraction to 0.8 for both train and validation splits:

```
>>> from evalne.evaluation.evaluator import SPEvaluator
>>> from evalne.evaluation.split import SPEvalSplit
>>> from evalne.utils import preprocess as pp
>>> # Load and preprocess a network
>>> G = pp.load_graph('./evalne/tests/data/sig_network.edgelist')
>>> G, _ = pp.prep_graph(G)
>>> # Create the required train/test split
>>> traintest_split = SPEvalSplit()
>>> _ = traintest_split.compute_splits(G, train_frac=0.8, split_alg='fast')
>>> # Check that the train test parameters are indeed the correct ones
>>> traintest_split.get_parameters()
{'split_id': 0, 'nw_name': 'test', 'split_alg': 'fast', 'train_frac': 0.8125}
>>> # Create the train/validation split from the train data computed in the␣
↪trintest_split
>>> # The graph used to initialize this split must, thus, be the train graph from␣
↪the traintest_split
>>> trainvalid_split = SPEvalSplit()
>>> _ = trainvalid_split.compute_splits(traintest_split.TG, train_frac=0.8, split_
↪alg='fast')
>>> # Initialize the SPEvaluator
>>> nee = SPEvaluator(traintest_split, trainvalid_split)
```

evaluate_baseline(*method*, *neighbourhood='in'*, *timeout=None*)

Evaluates the baseline method requested. Evaluation output is returned as a Results object. To evaluate the baselines on sign prediction we remove all negative edges from the train graph in traintest_split. For Katz neighbourhood='in' and neighbourhood='out' will return the same results corresponding to neighbourhood='in'. Execution time is contained in the results object. If the train/test split object used to initialize the evaluator does not contain test edges, the results object will only contain train results.

Parameters

- **method** (*string*) – A string indicating the name of any baseline from evalne.methods to evaluate.
- **neighbourhood** (*string, optional*) – A string indicating the 'in' or 'out' neighbourhood to be used for directed graphs. Default is 'in'.
- **timeout** (*float or None*) – A float indicating the maximum amount of time (in seconds) the evaluation can run for. If None, the evaluation is allowed to continue until

completion. Default is None.

Returns **results** – The evaluation results as a Results object.

Return type *Results*

Raises

- `TimeoutExpired` – If the execution does not finish within the allocated time.
- `TypeError` – If the Katz method call is incorrect.
- `ValueError` – If the heuristic selected does not exist.

See also:

evalne.utils.util.run_function() The low level function used to run a baseline with given timeout.

Examples

Evaluating the common neighbours heuristic with default parameters. We assume an evaluator (nee) has already been instantiated (see class examples):

```
>>> result = nee.evaluate_baseline(method='common_neighbours')
>>> # Print the results
>>> result.pretty_print()
Method: common_neighbours
Parameters:
[('split_id', 0), ('dim', 128), ('neighbourhood', 'in'), ('split_alg', 'fast
↪'),
('eval_time', 0.04459214210510254), ('nw_name', 'test'), ('train_frac', 0.
↪8125)]
Test scores:
tn = 71
[...]
```

Evaluating katz with beta=0.05 and timeout 60 seconds. We assume an evaluator (nee) has already been instantiated (see class examples):

```
>>> result = nee.evaluate_baseline(method='katz 0.05', timeout=60)
>>> # Print the results
>>> result.pretty_print()
Method: katz 0.05
Parameters:
[('split_id', 0), ('dim', 128), ('neighbourhood', 'in'), ('split_alg', 'fast
↪'),
('eval_time', 0.1246330738067627), ('nw_name', 'test'), ('train_frac', 0.
↪8125)]
Test scores:
tn = 120
[...]
```

evalne.evaluation.pipeline module

class evalne.evaluation.pipeline.**EvalSetup**(*configpath*, *run_checks=True*)

Bases: `object`

Class that acts as a wrapper for the EvalNE .ini configuration files. Options (or variables) in the .ini files are exposed as class properties and basic input checks are performed.

Parameters

- **configpath** (*string*) – The path of the .ini configuration file.
- **run_checks** (*bool, optional*) – Toggles .ini file parameter checks. Default is True.

comments
Returns a list of strings, the characters denoting comments in the network files.

curves
Returns a string indicating the curves to provide as output.

del_selfloops
Returns a bool, delete or not self loops in the network.

delimiter
Returns a string indicating the delimiter to be used when writing the preprocessed graphs to a files.

directed
Returns a bool indicating if all the networks are directed or not.

edge_embedding_methods
{'average', 'hadamard', 'weighted_l1', 'weighted_l2'}

Type Returns a list of strings indicating the node-pair operators to use. Possible values

embed_dim
Returns an int indicating the dimensions of the embedding.

embtype_other
node embeddings (ne), edge embeddings (ee) or node similarities (e2e). Possible values: {'ne', 'ee', 'e2e'}.

Type Returns a list of strings indicating the method's output type

fe_ratio
Returns a float indicating the ratio of non-edges to edges for tr & te. The num_fe = fe_ratio * num_edges.

getboollist (*section*, *option*)
Reads a string option and returns it as a list of booleans. The input string is split by any kind of white space separator. Elements such as 'True', 'true', '1', 'yes', 'on' are mapped to True. Elements such as 'False', 'false', '0', 'no', 'off' are mapped to False.

Parameters

- **section** (*string*) – A config file section name.
- **option** (*string*) – A config file option name.

Returns list – A list of booleans.

Return type list

getlinelist (*section*, *option*)
Reads a string option and returns it as a list of strings split by new lines only.

Parameters

- **section** (*string*) – A config file section name.
- **option** (*string*) – A config file option name.

Returns list – A list of strings.

Return type list

getlist(*section*, *option*, *dtype*)
Reads a string option and returns it as a list of elements of the specified type. The input string is split by any kind of white space separator.

Parameters

- **section** (*string*) – A config file section name.
- **option** (*string*) – A config file option name.
- **dtype** (*primitive type*) – The desired type of the elements in the output list.

Returns list – A list of elements cast to the specified primitive type.

Return type list

getseplist(*section*, *option*)
Reads a string option containing several separators ('s', 't' and 'n') and returns it as a list of proper string separators (white space, tab or new line).

Parameters

- **section** (*string*) – A config file section name.
- **option** (*string*) – A config file option name.

Returns list – A list of strings.

Return type list

gettuneparams(*library*)
Reads a 'TUNE_PARAMS' option that contain parameters and their associated values (e.g. 'TUNE_PARAMS'). The method returns the option as a list of strings split by new lines. The list if filled with None if needed so the length is the same as the number of methods being evaluated.

Parameters library (*string*) – A string indicating if the openne or other 'TUNE_PARAMS' should be checked. Accepted values are: 'opne', 'other'.

Returns tune_params – A list of string or None containing parameters and their values.

Return type list

inpaths
Returns a list of strings indicating the paths to files containing the networks. A check is performed to ensure the paths exist.

input_delim_other
Returns a list of strings indicating the input delimiters expected the by each methods.

labelpaths
Returns a list of string indicating the paths where the node label files can be found. Required if task is 'nc'

lp_baselines
{'', 'random_prediction', 'common_neighbours', 'jaccard_coefficient', 'adamic_adar_index', 'preferential_attachment', 'resource_allocation_index', 'cosine_similarity', 'lhn_index', 'topological_overlap', 'katz', 'all_baselines'}

Type Returns a list of strings indicating the link prediction heuristics to evaluate. Possible values

lp_model
Returns an sklearn binary classifier used to predict links from node-pair embeddings.

lp_num_edge_splits
Returns an int indicating the number of repetitions for experiment with different train/test edge splits. Required if task is 'lp' or 'sp'. For 'nr' and 'nc' this value must be 1.

maximize
Returns a string indicating the score to maximize when performing model validation. Possible values for LP, SP and NR: {'auroc', 'f_score', 'precision', 'recall', 'accuracy', 'fallout', 'miss'}. Possible values for NC: {'f1_micro', 'f1_macro', 'f1_weighted'}

methods_opne
Returns a list of strings indicating the command line calls to perform in order to evaluate each method.

methods_other
Returns a list of strings indicating the command line calls to perform in order to evaluate each method.

names
Returns a list of strings indicating the names of the networks to be evaluated.

names_opne
Returns a list of strings indicating the names of methods from OpenNE to be evaluated. In the same order as METHODS_OPNE.

names_other
Returns a list of strings indicating the names of any other methods not from OpenNE to be evaluated. In the same order as METHODS_OTHER.

nc_node_fracs
Returns a list of float indicating the fractions of train labels to use when evaluating NC. Required if task is 'nc'.

nc_num_node_splits
Returns an int indicating the number of repetitions for NC experiments with different train/test node splits. Required if task is 'nc'.

neighbourhood
Returns a list of string indicating, for directed graphs, if the in or the out neighbourhood should be used. Possible values: {'', 'in', 'out'}

nr_edge_samp_frac
Returns a float indicating the fraction of all possible node pairs to sample and compute precision@k for when evaluating NR. Required if task is 'nr'.

output_delim_other
Returns a list of strings indicating the delimiter used by each method in the output file (when writing node embeddings, edge embeddings or predictions).

output_format_other
Returns

owa
Returns a bool, indicating if the open world (True) or the closed world assumption (False) for non-edges should be used.

precatk_vals
Returns a list of int indicating the values of k for which to provide the precision at k.

relabel
Returns a bool, relabel or not the network nodes to 0...N (required for methods such as PRUNE)

save_prep_nw
Returns a bool if the preprocessed graph should be stored or not.

scores
{'', '%(maximize)s', 'all'}

Type Returns a string indicating the score to be reported in the output file. Possible values

seed
{'', 'None', any_int}

Type Returns and int or None indicating the random seed to use in the experiments. Possible values

separators
Returns a list of strings indicating the separators used in the network files.

split_alg
Returns a string indicating the algorithm to use for splitting edges in train/test, train/validation sets. Possible values: {'spanning_tree', 'random', 'naive', 'fast', 'timestamp'}.

task
Returns a string indicating the task to evaluate i.e. link prediction (LP), sign prediction (SP), network reconstruction (NR) or node classification (NC). Possible values: {'lp', 'sp', 'nr', 'nc'}

timeout
Returns a float indicating the maximum execution time in seconds (or None) for each method including hyperparameter tuning.

traintest_frac
Returns a float indicating the fraction of total edges to use for training and validation. The rest should be used for testing.

trainvalid_frac
Returns a float indicating the fraction of train-validation edges to use for training. The rest should be used for validation.

tune_params_opne
Returns a list of strings indicating the parameters of methods from OpenNE to be tuned by the library and values to try.

tune_params_other
Returns a list of strings indicating the parameters to be tuned by the library.

verbose
Returns a bool indicating the verbosity level of the execution.

write_dir_other
Returns a list of bool indicating if training graphs should be given as input to methods with both edge dir. (True) or one (False).

write_stats
Returns a bool, write or not common graph statistics as header in the preprocessed network file.

write_weights_other
Returns a list of bool indicating if training graphs should be given as input to methods weighted (True) or unweighted (False).

evalne.evaluation.score module

class evalne.evaluation.score.**NCResults**(*method*, *params*, *train_pred*, *train_labels*, *test_pred=None*, *test_labels=None*)

Bases: object

Class that encapsulates the train and test predictions of one method on a specific network and set of parameters. The train and test predictions are stored as NCScores objects. Functions for plotting, printing and saving to files the train and test scores are provided. Supports multi-label classification.

Parameters

- **method** (*string*) – A string representing the name of the method associated with these results.
- **params** (*dict*) – A dictionary of parameters used to obtain these results. Includes wall clock time of method evaluation.
- **train_pred** (*ndarray*) – An array containing the train predictions.
- **train_labels** (*ndarray*) – An array containing the train labels.
- **test_pred** (*ndarray, optional*) – An array containing the test predictions. Default is None.
- **test_labels** (*ndarray, optional*) – An array containing the test labels. Default is None.

Variables

- **method** (*string*) – A string representing the name of the method associated with these results.
- **params** (*dict*) – A dictionary of parameters used to obtain these results. Includes wall clock time of method evaluation.
- **train_scores** (Scores) – An NCScores object containing train scores.
- **test_scores** (Scores, *optional*) – An NCScores object containing test scores. Default is None.

get_all (*results='auto'*, *precatk_vals=None*)

Returns the names of all performance metrics that can be computed from train or test predictions and their associated values. These metrics are: 'f1_micro', 'f1_macro', 'f1_weighted'.

Parameters

- **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).
- **precatk_vals** (*None, optional*) – Not used.

Raises ValueError – If test results are requested but not initialized in constructor.

pretty_print (*results='auto'*)

Prints to screen the method name, execution parameters, and all available performance metrics (for train or test predictions).

Parameters **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).

Raises ValueError – If test results are requested but not initialized in constructor.

See also:

get_all() Describes all the performance metrics that can be computed from train or test predictions.

save(*filename*, *results='auto'*)
Writes the method name, execution parameters, and all available performance metrics (for train or test predictions) to a file.

Parameters

- **filename** (*string or file*) – A file or filename where to store the output.
- **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).

Raises ValueError – If test results are required but not initialized in constructor.

See also:

get_all() Describes all the performance metrics that can be computed from train or test predictions.

save_predictions(*filename*, *results='auto'*)
Writes the method name, execution parameters, and the train or test predictions to a file.

Parameters

- **filename** (*string or file*) – A file or filename where to store the output.
- **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).

Raises ValueError – If test results are required but not initialized in constructor.

class evalne.evaluation.score.**NCScores**(*y_true*, *y_pred*)
Bases: object

Class that encapsulates train or test predictions and exposes methods to compute different performance metrics. Supports multi-label classification.

Parameters

- **y_true** (*ndarray*) – An array containing the true labels.
- **y_pred** (*ndarray*) – An array containing the predictions.

Variables

- **y_true** (*ndarray*) – An array containing the true labels.
- **y_pred** (*ndarray*) – An array containing the predictions.

f1_macro()
Computes the f1 score for each label, and finds their unweighted average. This metric does not take label imbalance into account.

Returns **f1_macro** – The f1 macro score.

Return type float

f1_micro()
Computes the f1 score globally for all labels (i.e. sums the tp for all classes and divides by the sum of all tp+fp).

Returns **f1_micro** – The f1 micro score.

Return type float

f1_weighted()
Computes the f1 score for each label, and finds their average, weighted by support (the number of true instances for each label).

Returns **f1_weighted** – The weighted f1 score.

Return type float

class evalne.evaluation.score.**Results**(*method*, *params*, *train_pred*, *train_labels*, *test_pred=None*, *test_labels=None*, *label_binarizer=LogisticRegression(solver='liblinear')*)

Bases: object

Class that encapsulates the train and test predictions of one method on a specific network and set of parameters. The train and test predictions are stored as Scores objects. Functions for plotting, printing and saving to files the train and test scores are provided. Supports binary classification only.

Parameters

- **method** (*string*) – A string representing the name of the method associated with these results.
- **params** (*dict*) – A dictionary of parameters used to obtain these results. Includes wall clock time of method evaluation.
- **train_pred** (*ndarray*) – An array containing the train predictions.
- **train_labels** (*ndarray*) – An array containing the train labels.
- **test_pred** (*ndarray, optional*) – An array containing the test predictions. Default is None.
- **test_labels** (*ndarray, optional*) – An array containing the test labels. Default is None.
- **label_binarizer** (*string or Sklearn binary classifier, optional*) – If the predictions returned by the model are not binary, this parameter indicates how these binary predictions should be computed in order to be able to provide metrics such as the confusion matrix. Any Sklear binary classifier can be used or the keyword 'median' which will used the prediction medians as binarization thresholds. Default is LogisticRegression(solver='liblinear')

Variables

- **method** (*string*) – A string representing the name of the method associated with these results.
- **params** (*dict*) – A dictionary of parameters used to obtain these results. Includes wall clock time of method evaluation.
- **binary_preds** (*bool*) – A bool indicating if the train and test predictions are binary or not.
- **train_scores** (Scores) – A Scores object containing train scores.
- **test_scores** (Scores, *optional*) – A Scores object containing test scores. Default is None.
- **label_binarizer** (*string or Sklearn binary classifier, optional*) – If the predictions returned by the model are not binary, this parameter indicates how these binary predictions should be computed in order to be able to provide metrics such as the confusion matrix. By default, the method binarizes the predictions such that their accuracy is maximised. Any Sklearn binary classifier can be used or the keyword

'median' which will used the prediction medians as binarization thresholds. Default is LogisticRegression(solver='liblinear')

Raises `AttributeError` – If the label binarizer is set to an incorrect value.

get_all (*results='auto'*, *precatk_vals=None*)
Returns the names of all performance metrics that can be computed from train or test predictions and their associated values. These metrics are: 'tn', 'fp', 'fn', 'tp', 'auroc', 'average_precision', 'precision', 'precisionatk', 'recall', 'fallout', 'miss', 'accuracy' and 'f_score'.

Parameters

- **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).
- **precatk_vals** (*list of int or None, optional*) – The values for which the precision at k should be computed. Default is None.

Raises `ValueError` – If test results are requested but not initialized in constructor.

plot (*filename=None*, *results='auto'*, *curve='all'*)
Plots PR or ROC curves of the train or test predictions. If a filename is provided, the method will store the plot in pdf format to a file named <filename>+'_PR.pdf' or <filename>+'_ROC.pdf'.

Parameters

- **filename** (*string, optional*) – A string indicating the path and name of the file where to store the plot. If None, the plots are only shown on screen. Default is None.
- **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).
- **curve** (*string, optional*) – Can be one of 'all', 'pr' or 'roc'. Default is 'all' (generates both curves).

Raises `ValueError` – If test results are requested but not initialized in constructor.

pretty_print (*results='auto'*, *precatk_vals=None*)
Prints to screen the method name, execution parameters, and all available performance metrics (for train or test predictions).

Parameters

- **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).
- **precatk_vals** (*list of int or None, optional*) – The values for which the precision at k should be computed. Default is None.

Raises `ValueError` – If test results are requested but not initialized in constructor.

See also:

get_all() Describes all the performance metrics that can be computed from train or test predictions.

save (*filename*, *results='auto'*, *precatk_vals=None*)
Writes the method name, execution parameters, and all available performance metrics (for train or test predictions) to a file.

Parameters

- **filename** (*string or file*) – A file or filename where to store the output.
- **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).
- **precatk_vals** (*list of int or None, optional*) – The values for which the precision at k should be computed. Default is None.

Raises ValueError – If test results are required but not initialized in constructor.

See also:

get_all() Describes all the performance metrics that can be computed from train or test predictions.

save_predictions(*filename*, *results='auto'*)

Writes the method name, execution parameters, and the train or test predictions and corresponding labels to a file.

Parameters

- **filename** (*string or file*) – A file or filename where to store the output.
- **results** (*string, optional*) – A string indicating if the 'train' or 'test' predictions should be used. Default is 'auto' (selects 'test' if test predictions are logged and 'train' otherwise).

Raises ValueError – If test results are required but not initialized in constructor.

class evalne.evaluation.score.**Scores**(*y_true*, *y_pred*, *y_bin*)

Bases: object

Class that encapsulates train or test predictions and exposes methods to compute different performance metrics. Supports binary classification only.

Parameters

- **y_true** (*ndarray*) – An array containing the true labels.
- **y_pred** (*ndarray*) – An array containing the predictions.
- **y_bin** (*ndarray*) – An array containing binarized predictions.

Variables

- **y_true** (*ndarray*) – An array containing the true labels.
- **y_pred** (*ndarray*) – An array containing the predictions.
- **y_bin** (*ndarray*) – An array containing binarized predictions.
- **tn** (*float*) – The number of true negative in prediction.
- **fp** (*float*) – The number of false positives in prediction.
- **fn** (*float*) – The number of false negatives in prediction.
- **tp** (*float*) – The number of true positives in prediction.

accuracy()

Computes the accuracy score.

Returns **accuracy** – The prediction accuracy score.

Return type float

auroc()
Computes the Area Under the Receiver Operating Characteristic Curve (ROC AUC).

Returns **auroc** – The prediction auroc score.

Return type float

Notes

Throws a warning if class imbalance is detected.

average_precision()
Computes the average precision score.

Returns **avgprec** – The average precision score.

Return type float

f_score(*beta=1*)
Computes the F-score as the weighted harmonic mean of precision and recall.

Parameters **beta** (*float, optional*) – Allows to assign more weight to precision or recall. If beta > 1, recall is emphasized over precision. If beta < 1, precision is emphasized over recall.

Returns **f_score** – The prediction f_score.

Return type float

Notes

The generalized form is used, where P and R represent precision and recall, respectively:

$$F = (\beta^2 + 1) \cdot P \cdot R / (\beta^2 \cdot P + R)$$
$$F = (\beta^2 + 1) \cdot tp / ((\beta^2 + 1) \cdot tp + \beta^2 \cdot fn + fp)$$

fallout()
Computes the fallout in prediction.

Returns **fallout** – The prediction fallout score.

Return type float

miss()
Computes the miss in prediction.

Returns **miss** – The prediction miss score.

Return type float

precision()
Computes the precision in prediction.

Returns **precision** – The prediction precision score.

Return type float

precisionatk(*k=100*)
Computes the precision at k score.

Parameters **k** (*int, optional*) – The k value for which to compute the precision score. Default is 100.

Returns **precisionatk** – The prediction precision score for value k.

Return type float

recall()
Computes the recall in prediction.

Returns **recall** – The prediction recall score.

Return type float

class evalne.evaluation.score.**Scoresheet**(*tr_te='test'*, *precatk_vals=None*)
Bases: object

Class that simplifies the logging and management of evaluation results and execution times. Functions for logging, plotting and writing the results to files are provided. The Scoresheet does not log the complete train or test model predictions.

Parameters

- **tr_te** (*string, optional*) – A string indicating if the 'train' or 'test' results should be stored. Default is 'test'.
- **precatk_vals** (*list of int or None, optional*) – The values for which the precision at k should be computed. Default is None.

get_latex(*metric='auroc'*)
Returns a view of the Scoresheet as a Latex table for the specified metric. The columns of the table represent different networks and the rows different methods. If multiple Results for the same network/method combination were logged (multiple repetitions of the experiment), the average is returned.

Parameters **metric** (*string, optional*) – Can be one of 'tn', 'fp', 'fn', 'tp', 'auroc', 'average_precision', 'precision', 'recall', 'fallout', 'miss', 'accuracy', 'f_score', 'eval_time' or 'edge_embed_method'. Default is 'auroc'.

Returns **latex_table** – A latex table as a string.

Return type string

get_pandas_df(*metric='auroc'*, *repeat=None*)
Returns a view of the Scoresheet as a pandas DataFrame for the specified metric. The columns of the DataFrame represent different networks and the rows different methods. If multiple Results for the same network/method combination were logged (multiple repetitions of the experiment), one can select any of these repeats or get the average over all.

Parameters

- **metric** (*string, optional*) – Can be one of 'tn', 'fp', 'fn', 'tp', 'auroc', 'average_precision', 'precision', 'recall', 'fallout', 'miss', 'accuracy', 'f_score', 'eval_time' or 'edge_embed_method'. Default is 'auroc'.
- **repeat** (*int, optional*) – An int indicating the experiment repeat for which the results should be returned. If not indicated, the average over all repeats will be computed and returned. Default is None (computes average over repeats).

Returns **df** – A pandas DataFrame view of the Scoresheet for the specified metric.

Return type DataFrame

Raises ValueError – If the requested metric does not exist. If the Scoresheet is empty so a DataFrame can not be generated.

Notes

For uncountable 'metrics' such as the node pair embedding operator (i.e 'edge_embed_method'), avg returns the most frequent item in the vector.

Examples

Read a scoresheet and get the auroc scores as a pandas DataFrame

```
>>> scores = pickle.load(open('lp_eval_1207_1638/eval.pkl', 'rb'))
>>> df = scores.get_pandas_df()
>>> df
                    Network_1 Network_2
katz                   0.8203    0.8288
common_neighbours      0.3787    0.3841
jaccard_coefficient    0.3787    0.3841
```

Read a scoresheet and get the f scores of the first repetition of the experiment

```
>>> scores = pickle.load(open('lp_eval_1207_1638/eval.pkl', 'rb'))
>>> df = scores.get_pandas_df('f_score', repeat=0)
>>> df
                    Network_1 Network_2
katz                        0         0
common_neighbours      0.7272    0.7276
jaccard_coefficient    0.7265    0.7268
```

log_results (*results*)

Logs in the Scoresheet all the performance metrics (and execution time) extracted from the input Results object or list of Results objects. Multiple Results for the same method on the same network can be provided and will all be stored (these are assumed to correspond to different repetitions of the experiment).

Parameters **results** (*Results or list of Results*) – The Results object or objects to be logged in the Scoresheet.

Examples

Evaluate the common neighbours baseline and log the train and test results:

```
>>> tr_scores = Scoresheet(tr_te='train')
>>> te_scores = Scoresheet(tr_te='test')
>>> result = nee.evaluate_baseline(method='common_neighbours')
>>> tr_scores.log_results(result)
>>> te_scores.log_results(result)
```

print_tabular (*metric='auroc'*)

Prints a tabular view of the Scoresheet for the specified metric. The columns of the table represent different networks and the rows different methods. If multiple Results for the same network/method combination were logged (multiple repetitions of the experiment), the average is showed.

Parameters **metric** (*string, optional*) – Can be one of 'tn', 'fp', 'fn', 'tp', 'auroc', 'average_precision', 'precision', 'recall', 'fallout', 'miss', 'accuracy', 'f_score', 'eval_time' or 'edge_embed_method'. Default is 'auroc'.

Examples

Read a scoresheet and get the average execution times over all experiment repeats as tabular output:

```
>>> scores = pickle.load(open('lp_eval_1207_1638/eval.pkl', 'rb'))
>>> scores.print_tabular('eval_time')
                    Network_1 Network_2
katz                   0.0350    0.0355
common_neighbours      0.0674    0.0676
jaccard_coefficient    0.6185    0.6693
```

write_all (*filename*, *repeats='avg'*)

Writes for all networks, methods and performance metrics the corresponding values to a file. If multiple Results for the same network/method combination were logged (multiple repetitions of the experiment), the method can return the average or all logged values.

Parameters

- **filename** (*string*) – A file where to store the results.
- **repeats** (*string, optional*) – Can be one of 'all', 'avg'. Default is 'avg'.

Notes

For uncountable 'metrics' such as the node pair embedding operator (i.e 'edge_embed_method'), avg returns the most frequent item in the vector.

Examples

Read a scoresheet and write all metrics to a file with repeats='avg':

```
>>> scores = pickle.load(open('lp_eval_1207_1638/eval.pkl', 'rb'))
>>> scores.write_all('./test.txt')
>>> print(open('test.txt', 'rb').read())
Network_1 Network
---------------------------
katz:
 tn:     684.0
 fp:     0.0
 fn:     684.0
 tp:     0.0
 auroc:          0.8203
...
```

Read a scoresheet and write all metrics to a file with repeats='all':

```
>>> scores = pickle.load(open('lp_eval_1207_1638/eval.pkl', 'rb'))
>>> scores.write_all('./test.txt', 'all')
>>> print(open('test.txt', 'rb').read())
Network_1 Network
---------------------------
katz:
 tn:     [684 684]
 fp:     [0 0]
 fn:     [684 684]
 tp:     [0 0]
```

(continues on next page)

(continued from previous page)

```
 auroc:          [0.8155 0.8252]
...
```

write_pickle (*filename*)
Writes a pickle representation of this object to a file.

Parameters **filename** (*string*) – A file where to store the pickle representation.

write_tabular (*filename*, *metric='auroc'*)
Writes a tabular view of the Scoresheet for the specified metric to a file. The columns of the table represent different networks and the rows different methods. If multiple Results for the same network/method combination were logged (multiple repetitions of the experiment), the average is used.

Parameters

- **filename** (*string*) – A file where to store the results.
- **metric** (*string, optional*) – Can be one of 'tn', 'fp', 'fn', 'tp', 'auroc', 'average_precision', 'precision', 'recall', 'fallout', 'miss', 'accuracy', 'f_score' or 'eval_time'. Default is 'auroc'.

evalne.evaluation.split module

class evalne.evaluation.split.**BaseEvalSplit**
Bases: object

Base class that provides a high level interface for managing/computing sets of train and test edges and non-edges for LP, SP and NR tasks. The class exposes the train edges and non-edges through the *train_edges* property and the test edges and non-edges through the *test_edges* property. Parameters used to compute these sets are also made available.

TG
A NetworkX graph or digraph to be used for training the embedding methods. For LP this should be the graph spanned by all train edges, for SP the graph spanned by the positive and negative train edges (with signs as edge weights) and for NR the entire graph being evaluated.

get_data ()
Returns the sets of train and test node pairs and label vectors.

Returns

- **train_edges** (*set*) – Set of all train edges and non-edges.
- **test_edges** (*set*) – Set of all test edges and non-edges.
- **train_labels** (*list*) – A list of labels indicating if each train node-pair is an edge or non-edge (1 or 0).
- **test_labels** (*list*) – A list of labels indicating if each test node-pair is an edge or non-edge (1 or 0).

get_parameters ()
Returns the class properties except the sets of train and test node pairs, labels and train graph.

Returns **parameters** – The parameters used when computing this split as a dictionary of parameters and values.

Return type dict

nw_name
A string indicating the name of the dataset used to generate the sets of edges.

save_tr_graph(*output_path*, *delimiter*, *write_stats=False*, *write_weights=False*, *write_dir=True*)
Saves the TG graph to a file.

Parameters

- **output_path** (*file or string*) – File or filename to write. If a file is provided, it must be opened in 'wb' mode.
- **delimiter** (*string, optional*) – The string used to separate values. Default is ','.
- **write_stats** (*bool, optional*) – Adds basic graph statistics to the file as a header or not. Default is True.
- **write_weights** (*bool, optional*) – If True data will be stored as weighted edge-list i.e. triplets (src, dst, weight), otherwise, as regular (src, dst) pairs. For unweighted graphs, setting this parameter to True will add weight 1 to all edges. Default is False.
- **write_dir** (*bool, optional*) – This parameter is only relevant for undirected graphs. If True, it forces the method to write both edge directions in the file i.e. (src, dst) and (dst, src). If False, only one direction is stored. Default is True.

See also:

evalne.utils.preprocess.save_graph()

split_alg
A string indicating the algorithm used to split edges in train and test sets.

split_id
An int used as an ID for this particular train/test split.

store_edgelists(*train_path*, *test_path*)
Writes the sets of train and test node pairs to files with the specified names.

Parameters

- **train_path** (*string*) – Indicates the path where the train data will be stored.
- **test_path** (*string*) – Indicates the path where the test data will be stored.

See also:

evalne.utils.split_train_test.store_edgelists()

test_edges
The set of test node pairs.

test_labels
A list of test node-pair labels. Labels can be either 0 or 1 and denote non-edges and edges, respectively (for SP they denote negative and positive links, respectively).

train_edges
The set of training node pairs.

train_frac
A float indicating the fraction of train edges out of all train and test edges.

train_labels
A list of train node-pair labels. Labels can be either 0 or 1 and denote non-edges and edges, respectively (for SP they denote negative and positive links, respectively).

class evalne.evaluation.split.**EvalSplit**
Bases: *evalne.evaluation.split.LPEvalSplit*

Deprecated and will be removed in v0.4.0. Use LPEvalSplit instead.

read_splits(*filename*, *split_id*, *directed=False*, *nw_name='test'*, *verbose=False*)
Reads the train and test edges and non-edges from files and initializes the class attributes.

Parameters

- **filename** (*string*) – The filename shared by all edge splits as given by the 'store_train_test_splits' method
- **split_id** (*int*) – The ID of the edge splits to read. As provided by the 'store_train_test_splits' method
- **directed** (*bool, optional*) – True if the splits correspond to a directed graph, false otherwise. Default is False.
- **nw_name** (*string, optional*) – A string indicating the name of the dataset from which this split was generated. This is required in order to keep track of the evaluation results. Default is *test*.
- **verbose** (*bool, optional*) – If True print progress info. Default is False.

See also:

evalne.utils.preprocess.read_train_test() The low level function used for reading the sets of edges and non-edges.

evalne.utils.split_train_test.store_train_test_splits() The files in the provided input path are expected to follow the naming convention of this function.

class evalne.evaluation.split.**LPEvalSplit**
Bases: *evalne.evaluation.split.BaseEvalSplit*

Class that provides a high level interface for managing/computing sets of train and test edges and non-edges for LP tasks. The class exposes the train edges and non-edges through the *train_edges* property and the test edges and non-edges through the *test_edges* property. Parameters used to compute these sets are also made available.

Notes

In link prediction the aim is to predict, given a set of node pairs, if they should be connected or not. This is generally solved as a binary classification task. For training the binary classifier, we sample a set of edges as well as a set of unconnected node pairs. We then compute the node-pair embeddings of this training data. We use the node-pair embeddings together with the corresponding labels (0 for non-edges and 1 for edges) to train the classifier. Finally, the performance is evaluated on the test data (the remaining edges not used in training plus another set of randomly selected non-edges).

compute_splits(*G*, *nw_name='test'*, *train_frac=0.51*, *split_alg='spanning_tree'*, *owa=True*, *fe_ratio=1*, *split_id=0*, *verbose=False*)
Computes sets of train and test edges and non-edges according to the given input parameters and initializes the class attributes.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph to compute the train test split from.
- **nw_name** (*string, optional*) – A string indicating the name of the dataset from which this split was generated. This is required in order to keep track of the evaluation results. Default is 'test'.

- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.
- **split_alg** (*string, optional*) – A string indicating the algorithm to use for generating the train/test splits. Options are *spanning_tree*, *random*, *naive*, *fast* and *timestamp*. Default is *spanning_tree*.
- **owa** (*bool, optional*) – Encodes the belief that the network should respect or not the open world assumption. Default is True. If owa=True, train non-edges are sampled from the train graph only and can overlap with test edges. If owa=False, train non-edges are sampled from the full graph and cannot overlap with test edges.
- **fe_ratio** (*float, optional*) – The ratio of non-edges to edges to sample. For fr_ratio > 0 and < 1 less non-edges than edges will be generated. For fe_edges > 1 more non-edges than edges will be generated. Default 1, same amounts.
- **split_id** (*int, optional*) – The id to be assigned to the train/test splits generated. Default is 0.
- **verbose** (*bool, optional*) – If True print progress info. Default is False.

Returns

- **train_E** (*set*) – The set of train edges
- **train_false_E** (*set*) – The set of train non-edges
- **test_E** (*set*) – The set of test edges
- **test_false_E** (*set*) – The set of test non-edges

Raises `ValueError` – If the edge split algorithm is unknown.

fe_ratio
A float indicating the ratio of non-edges to edges.

get_parameters()
Returns the class properties except the sets of train and test node pairs, labels and train graph.

Returns parameters – The parameters used when computing this split as a dictionary of parameters and values.

Return type dict

owa
A bool parameter indicating if the non-edges have been generated using the OWA (otherwise CWA).

set_splits(*train_E*, *train_E_false=None*, *test_E=None*, *test_E_false=None*, *directed=False*, *nw_name='test'*, *TG=None*, *split_id=0*, *split_alg='spanning_tree'*, *owa=True*, *verbose=False*)
Sets the class attributes to the provided input values. The input train edges and non-edges as well as the test edges and non-edges are respectively joined to form the *train_edges* and *test_edges* class attributes. Train and test labels are also inferred from the input data.

Parameters

- **train_E** (*set*) – Set of train edges.
- **train_E_false** (*set, optional*) – Set of train non-edges. Default is None.
- **test_E** (*set, optional*) – Set of test edges. Default is None.
- **test_E_false** (*set, optional*) – Set of test non-edges. Default is None.
- **directed** (*bool, optional*) – True if the splits correspond to a directed graph, false otherwise. Default is False.

- **nw_name** (*string, optional*) – A string indicating the name of the dataset from which this split was generated. This is required in order to keep track of the evaluation results. Default is *test*.
- **TG** (*graph, optional*) – A NetworkX graph or digraph containing all the train edges. If None, the graph will be generated from the set of train edges. Default is None.
- **split_id** (*int, optional*) – An ID that identifies this particular train/test split. Default is 0.
- **split_alg** (*string, optional*) – A string indicating the algorithm used to generate the train/test splits. Options are *spanning_tree*, *random*, *naive*, *fast* and *timestamp*. Default is *spanning_tree*.
- **owa** (*bool, optional*) – Encodes the belief that the network respects or not the open world assumption. Default is True. If owa=True, train non-edges are sampled from the train graph only and can overlap with test edges. If owa=False, train non-edges are sampled from the full graph and cannot overlap with test edges.
- **verbose** (*bool, optional*) – If True prints progress info. Default is False.

Raises ValueError – If the train edge set is not provided.

class evalne.evaluation.split.**NREvalSplit**

Bases: *evalne.evaluation.split.BaseEvalSplit*

Class that provides a high level interface for managing/computing sets of train edges and non-edges for NR tasks. The class exposes the train edges and non-edges through the *train_edges* property. Test edges are not used for NR and therefore the *test_edges* property will be left empty. Parameters used to compute these sets are also made available.

Notes

In network reconstruction the aim is to asses how well an embedding method captures the structure of a given graph. The embedding methods are trained on a complete input graph. Hyperparameter tuning is performed directly on this graph (overfitting is, in this case, expected and desired). The embeddings obtained are used to perform link predictions and their quality is evaluated. Checking the link predictions for all node pairs is generally unfeasible, therefore a subset of all node pairs in the input graph are selected for evaluation.

compute_splits(*G*, *nw_name='test'*, *samp_frac=0.01*, *split_id=0*, *verbose=False*)

Computes sets of train edges and non-edges by randomly sampling elements from the adjacency matrix of G and initializes the class attributes.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph to sample node pairs from.
- **nw_name** (*string, optional*) – A string indicating the name of the dataset from which this split was generated. This is required in order to keep track of the evaluation results. Default is 'test'.
- **samp_frac** (*float, optional*) – The fraction of node-pairs out of all possible ones to sample for NR evaluation. Default is 0.01 (1%).
- **split_id** (*int, optional*) – The id to be assigned to the train/test splits generated. Default is 0.
- **verbose** (*bool, optional*) – If True print progress info. Default is False.

Returns

- **train_E** (*set*) – The set of train edges.

- **train_false_E** (*set*) – The set of train non-edges.

Raises `ValueError` – If the edge split algorithm is unknown.

get_parameters()
Returns the class properties except the sets of train and test node pairs, labels and train graph.

Returns **parameters** – The parameters used when computing this split as a dictionary of parameters and values.

Return type dict

samp_frac
A float indicating the fraction of node pairs out of all possible ones sampled for NR evaluation.

set_splits(*TG*, *train_E*, *train_E_false=None*, *samp_frac=None*, *directed=False*, *nw_name='test'*, *split_id=0*, *verbose=False*)
Sets the class attributes to the provided input values. The input train edges and non-edges are joined to form the *train_edges* class attribute. Train labels are also inferred from the input data.

Parameters

- **TG** (*graph*) – A NetworkX graph or digraph, the complete network from which train_E and train_E_false were sampled.
- **train_E** (*set*) – Set of train edges.
- **train_E_false** (*set, optional*) – Set of train non-edges. Default is None.
- **samp_frac** (*float, optional*) – The fraction of node-pairs out of all possible ones sampled for NR evaluation. Default is None.
- **directed** (*bool, optional*) – True if the splits correspond to a directed graph, false otherwise. Default is False.
- **nw_name** (*string, optional*) – A string indicating the name of the dataset from which this split was generated. This is required in order to keep track of the evaluation results. Default is *test*.
- **split_id** (*int, optional*) – An ID that identifies this particular train/test split. Default is 0.
- **verbose** (*bool, optional*) – If True prints progress info. Default is False.

Raises `ValueError` – If the train edge set is not provided.

class evalne.evaluation.split.**SPEvalSplit**
Bases: *evalne.evaluation.split.BaseEvalSplit*

Class that provides a high level interface for managing/computing sets of train and test positive and negative edges for SP tasks. The class exposes the train positive and negative edges through the *train_edges* property and the test positive and negative edges through the *test_edges* property. Parameters used to compute these sets are also made available.

Notes

In sign prediction the aim is to predict the sign (positive or negative) of given edges. The existence of the edges is assumed (i.e. we do not predict the sign of unconnected node pairs). Therefore, sign prediction is also a binary classification task similar to link prediction where, instead of predicting the existence of edges or not, we predict the signs for edges we know exist. Unlike for link prediction, in this case we do not need to perform negative sampling, since we already have both classes (the positively and the negatively connected node pairs).

compute_splits(*G*, *nw_name='test'*, *train_frac=0.51*, *split_alg='spanning_tree'*, *split_id=0*, *verbose=False*)

Computes sets of train and test positive and negative edges according to the given input parameters and initializes the class attributes.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph to compute the train test split from.
- **nw_name** (*string, optional*) – A string indicating the name of the dataset from which this split was generated. This is required in order to keep track of the evaluation results. Default is 'test'.
- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.
- **split_alg** (*string, optional*) – A string indicating the algorithm to use for generating the train/test splits. Options are *spanning_tree*, *random*, *naive*, *fast* and *timestamp*. Default is *spanning_tree*.
- **split_id** (*int, optional*) – The id to be assigned to the train/test splits generated. Default is 0.
- **verbose** (*bool, optional*) – If True print progress info. Default is False.

Returns

- **train_E** (*set*) – The set of train positive edges.
- **train_false_E** (*set*) – The set of train negative edges.
- **test_E** (*set*) – The set of test positive edges.
- **test_false_E** (*set*) – The set of test negative edges.

Raises ValueError – If the edge split algorithm is unknown.

set_splits(*train_E*, *train_E_false=None*, *test_E=None*, *test_E_false=None*, *directed=False*, *nw_name='test'*, *TG=None*, *split_id=0*, *split_alg='spanning_tree'*, *verbose=False*)

Sets the class attributes to the provided input values. The input train positive and negative edges as well as the test positive and negative edges are respectively joined to form the *train_edges* and *test_edges* class attributes. Train and test labels (0 or 1 representing negative and positive edges, respectively) are also inferred from the input data.

Parameters

- **train_E** (*set*) – Set of positive train edges.
- **train_E_false** (*set, optional*) – Set of negative train edges. Default is None.
- **test_E** (*set, optional*) – Set of positive test edges. Default is None.
- **test_E_false** (*set, optional*) – Set of negative test edges. Default is None.
- **directed** (*bool, optional*) – True if the splits correspond to a directed graph, false otherwise. Default is False.
- **nw_name** (*string, optional*) – A string indicating the name of the dataset from which this split was generated. This is required in order to keep track of the evaluation results. Default is *test*.
- **TG** (*graph, optional*) – A NetworkX graph or digraph containing all the train edges (positive and negative). If None, the graph will be generated from the sets of positive and negative train edges. Default is None.

- **split_id** (*int, optional*) – An ID that identifies this particular train/test split. Default is 0.
- **split_alg** (*string, optional*) – A string indicating the algorithm used to generate the train/test splits. Options are *spanning_tree*, *random*, *naive*, *fast* and *timestamp*. Default is *spanning_tree*.
- **verbose** (*bool, optional*) – If True prints progress info. Default is False.

Raises ValueError – If the train edge set is not provided.

Module contents

evalne.methods package

Submodules

evalne.methods.katz module

class evalne.methods.katz.**Katz**(*G*, *beta=0.005*)

Bases: object

Computes the exact katz similarity based on paths between nodes in the graph. Shorter paths will contribute more than longer ones. This contribution depends of the damping factor 'beta'. The exact katz score is computed using the adj matrix of the full graph.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph with nodes being consecutive integers starting at 0.
- **= float, optional** (*beta*) – The damping factor for the model. Default is 0.005.

Notes

The execution is based on dense matrices, so it may run out of memory.

get_params()

Returns a dictionary of model parameters.

Returns **params** – A dictionary of model parameters and their values.

Return type dict

predict(*ebunch*)

Computes the katz score for all node-pairs in ebunch.

Parameters **ebunch** (*iterable*) – An iterable of node-pairs for which to compute the katz score.

Returns An array containing the similarity scores.

Return type ndarray

save_sim_matrix(*filename*)

Stores the similarity matrix to a file with the given name.

Parameters **filename** (*string*) – The name and path of the file where the similarity matrix should be stored.

class evalne.methods.katz.**KatzApprox**(*G*, *beta=0.005*, *path_len=3*)
Bases: object

Computes the approximated katz similarity based on paths between nodes in the graph. Shorter paths will contribute more than longer ones. This contribution depends of the damping factor 'beta'. The approximated score is computed using all paths between nodes of length at most 'path_len'.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **beta** (*float, optional*) – The damping factor for the model. Default is 0.005.
- **path_len** (*int, optional*) – The maximum path length to consider between each pair of nodes. Default is 3.

Notes

The implementation follows the indication in [1]. It becomes extremely slow for large dense graphs.

References

fit_predict(*ebunch*)
Computes the katz score for all node-pairs in ebunch.

Parameters **ebunch** (*iterable*) – An iterable of node-pairs for which to compute the katz score.

Returns An array containing the similarity scores.

Return type ndarray

get_params()
Returns a dictionary of model parameters.

Returns **params** – A dictionary of model parameters and their values.

Return type dict

evalne.methods.similarity module

evalne.methods.similarity.**common_neighbours**(*G*, *ebunch=None*, *neighbourhood='in'*)
Computes the common neighbours similarity between all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs (see Notes for exact definitions).

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

Raises ValueError – If G is directed and neighbourhood is not one of 'in' or 'out'.

Notes

For undirected graphs the common neighbours similarity of nodes 'u' and 'v' is defined as:

$$|\Gamma(u) \cap \Gamma(v)|$$

For directed graphs we can consider either the in or the out-neighbourhoods, respectively:

$$|\Gamma_i(u) \cap \Gamma_i(v)|$$
$$|\Gamma_o(u) \cap \Gamma_o(v)|$$

evalne.methods.similarity.**jaccard_coefficient**(*G*, *ebunch=None*, *neighbourhood='in'*)

Computes the Jaccard coefficient between all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs (see Notes for exact definitions).

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

Raises ValueError – If G is directed and neighbourhood is not one of 'in' or 'out'.

Notes

For undirected graphs the Jaccard coefficient of nodes 'u' and 'v' is defined as:

$$|\Gamma(u) \cap \Gamma(v)|/|\Gamma(u) \cup \Gamma(v)|$$

For directed graphs we can consider either the in or the out-neighbourhoods, respectively:

$$\frac{|\Gamma_i(u) \cap \Gamma_i(v)|}{|\Gamma_i(u) \cup \Gamma_i(v)|}$$
$$\frac{|\Gamma_o(u) \cap \Gamma_o(v)|}{|\Gamma_o(u) \cup \Gamma_o(v)|}$$

evalne.methods.similarity.**cosine_similarity**(*G*, *ebunch=None*, *neighbourhood='in'*)

Computes the cosine similarity between all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs (see Notes for exact definitions).

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

Raises ValueError – If G is directed and neighbourhood is not one of 'in' or 'out'.

Notes

For undirected graphs the cosine similarity of nodes 'u' and 'v' is defined as:

$$\frac{|\Gamma(u) \cap \Gamma(v)|}{\sqrt{|\Gamma(u)||\Gamma(v)|}}$$

For directed graphs we can consider either the in or the out-neighbourhoods, respectively:

$$\frac{|\Gamma_i(u) \cap \Gamma_i(v)|}{\sqrt{|\Gamma_i(u)||\Gamma_i(v)|}}$$
$$\frac{|\Gamma_o(u) \cap \Gamma_o(v)|}{\sqrt{|\Gamma_o(u)||\Gamma_o(v)|}}$$

evalne.methods.similarity.**lhn_index**(*G*, *ebunch=None*, *neighbourhood='in'*)

Computes the Leicht-Holme-Newman index [1]_ between all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs (see Notes for exact definitions).

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

Raises ValueError – If G is directed and neighbourhood is not one of 'in' or 'out'.

Notes

For undirected graphs the Leicht-Holme-Newman index of nodes 'u' and 'v' is defined as:

$$\frac{|\Gamma(u) \cap \Gamma(v)|}{|\Gamma(u)||\Gamma(v)|}$$

For directed graphs we can consider either the in or the out-neighbourhoods, respectively:

$$\frac{|\Gamma_i(u) \cap \Gamma_i(v)|}{|\Gamma_i(u)||\Gamma_i(v)|}$$
$$\frac{|\Gamma_o(u) \cap \Gamma_o(v)|}{|\Gamma_o(u)||\Gamma_o(v)|}$$

References

evalne.methods.similarity.**topological_overlap**(*G*, *ebunch=None*, *neighbourhood='in'*)

Computes the topological overlap[2] between all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs (see Notes for exact definitions).

Parameters

[2] Ravasz, E., Somera, A. L., Mongru, D. A., Oltvai, Z. N., & Barabási, A. L. (2002). "Hierarchical organization of modularity in metabolic networks." Science, 297(5586), 1551-1555.

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

Raises ValueError – If G is directed and neighbourhood is not one of 'in' or 'out'.

Notes

For undirected graphs the topological overlap of nodes 'u' and 'v' is defined as:

$$\frac{|\Gamma(u) \cap \Gamma(v)|}{min(|\Gamma(u)|, |\Gamma(v)|)}$$

For directed graphs we can consider either the in or the out-neighbourhoods, respectively:

$$\frac{|\Gamma_i(u) \cap \Gamma_i(v)|}{min(|\Gamma_i(u)|, |\Gamma_i(v)|)}$$
$$\frac{|\Gamma_o(u) \cap \Gamma_o(v)|}{min(|\Gamma_o(u)|, |\Gamma_o(v)|)}$$

References

evalne.methods.similarity.**adamic_adar_index**(*G*, *ebunch=None*, *neighbourhood='in'*)

Computes the Adamic-Adar index between all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs (see Notes for exact definitions).

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

Raises ValueError – If G is directed and neighbourhood is not one of 'in' or 'out'.

Notes

For undirected graphs the Adamic-Adar index of nodes 'u' and 'v' is defined as:

$$\sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{\log |\Gamma(w)|}$$

For directed graphs we can consider either the in or the out-neighbourhoods, respectively:

$$\sum_{w \in \Gamma_i(u) \cap \Gamma_i(v)} \frac{1}{\log |\Gamma_i(w)|}$$
$$\sum_{w \in \Gamma_o(u) \cap \Gamma_o(v)} \frac{1}{\log |\Gamma_o(w)|}$$

evalne.methods.similarity.**resource_allocation_index**(*G*, *ebunch=None*, *neighbourhood='in'*)

Computes the resource allocation index between all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs (see Notes for exact definitions).

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

Raises ValueError – If G is directed and neighbourhood is not one of 'in' or 'out'.

Notes

For undirected graphs the resource allocation index of nodes 'u' and 'v' is defined as:

$$\sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{|\Gamma(w)|}$$

For directed graphs we can consider either the in or the out-neighbourhoods, respectively:

$$\sum_{w \in \Gamma_i(u) \cap \Gamma_i(v)} \frac{1}{|\Gamma_i(w)|}$$
$$\sum_{w \in \Gamma_o(u) \cap \Gamma_o(v)} \frac{1}{|\Gamma_o(w)|}$$

evalne.methods.similarity.**preferential_attachment**(*G*, *ebunch=None*, *neighbourhood='in'*)

Computes the preferential attachment score between all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs (see Notes for exact definitions).

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

Raises `ValueError` – If G is directed and neighbourhood is not one of 'in' or 'out'.

Notes

For undirected graphs the preferential attachment score of nodes 'u' and 'v' is defined as:

$$|\Gamma(u)||\Gamma(v)|$$

For directed graphs we can consider either the in or the out-neighbourhoods, respectively:

$$|\Gamma_i(u)||\Gamma_i(v)|$$
$$|\Gamma_o(u)||\Gamma_o(v)|$$

`evalne.methods.similarity.`**`random_prediction`**(*G*, *ebunch=None*, *neighbourhood='in'*)

Returns a float drawn uniformly at random from the interval (0.0, 1.0] for all node pairs in ebunch; or all nodes in G, if ebunch is None. Can be computed for directed and undirected graphs.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – Not used.

Returns **sim** – A list of node-pair similarities in the same order as ebunch.

Return type list

`evalne.methods.similarity.`**`all_baselines`**(*G*, *ebunch=None*, *neighbourhood='in'*)

Computes a 5-dimensional embedding for all node pairs in ebunch; or all nodes in G, if ebunch is None. Each of the 5 dimensions correspond to the similarity between the nodes as computed by a different function (i.e. CN, JC, AA, RAI and PA). Can be computed for directed and undirected graphs.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **ebunch** (*iterable, optional*) – An iterable of node pairs. If None, all edges in G will be used. Default is None.
- **neighbourhood** (*string, optional*) – For directed graphs only. Determines if the in or the out-neighbourhood of nodes should be used. Default is 'in'.

Returns **emb** – Column vector containing node-pair embeddings as rows.

Return type ndarray

Raises `ValueError` – If G is directed and neighbourhood is not one of 'in' or 'out'.

Module contents

evalne.utils package

Submodules

evalne.utils.preprocess module

evalne.utils.preprocess.**get_redges_false**(*G*, *output_path=None*)

For directed graphs returns a list of all non-edges for which the opposite edge exists in the graph. E.g. returns all pairs of non-edges (a -> b) such that (b -> a) exists in the graph.

Parameters

- **G** (*graph*) – A NetworkX digraph.
- **output_path** (*string*) – A path or file where to store the results.

Returns **redges_false** – A set of node pairs respecting the mentioned property.

Return type set

evalne.utils.preprocess.**get_stats**(*G*, *output_path=None*, *all_stats=False*)

Prints or stores some basic statistics about the graph. If an output path is provided the results are written in said file.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **output_path** (*file or string, optional*) – File or filename to write. Default is None
- **all_stats** (*bool, optional*) – Sets if all stats or a small subset of them should be shown. Computing all stats can be very slow. Default is False.

evalne.utils.preprocess.**infer_header**(*input_path*, *expected_lines*, *method_name=None*)

Method that infers the length of the header of a given file from the number of lines and of expected lines.

Parameters

- **input_path** (*file or string*) – File or filename to read.
- **expected_lines** (*int*) – Number of expected lines in the input file.
- **method_name** (*string, optional*) – A string indicating the name of the method being evaluated. If provided will be used when logging an error. Default is None.

Returns **header_len** – The length of the header.

Return type int

Raises ValueError – If more lines are expected than those in the file.

evalne.utils.preprocess.**load_graph**(*input_path*, *delimiter=' '*, *comments='#'*, *directed=False*, *datatype=<class 'float'>*)

Reads a directed or undirected edgelist and returns a graph. If the edgelist is weighted the graph will maintain those weights. Nodes are read as int and weights as float.

Parameters

- **input_path** (*file or string*) – File or filename to read.
- **delimiter** (*string, optional*) – The string used to separate values in the input file. Default is ‘,’.
- **comments** (*string, optional*) – The character used to indicate the start of a comment. Default is ‘#’.
- **directed** (*bool*) – Indicates if the method should return a graph or a digraph.

- **datatype** (*int or float, optional*) – The type of the graph weights. Default is float.

Returns **G** – A NetworkX graph or digraph.

Return type graph

evalne.utils.preprocess.**prep_graph**(*G*, *relabel=True*, *del_self_loops=True*, *maincc=True*)

Preprocess a graph according to the input parameters.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **relabel** (*bool, optional*) – Determines if the nodes should be relabeled with consecutive integers 0...N. Default is True.
- **del_self_loops** (*bool, optional*) – Determines if self loops should be deleted. Default is True.
- **maincc** (*bool, optional*) – Determines if the graphs (digraphs) should be restricted to the main (weakly) connected component or not. Default is True.

Returns

- **G** (*graph*) – A preprocessed NetworkX graph or digraph.
- **Ids** (*list of tuples*) – A list of (OldNodeID, NewNodeID). Returns None if relabel=False.

Notes

For some methods trying to embed graphs with several connected components might result in inconsistent behaviours. This is the main reason for setting maincc=True.

evalne.utils.preprocess.**prune_nodes**(*G*, *threshold*)

Removes all nodes from the graph whose degree is strictly smaller that the threshold. This can result in a disconnected graph.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **threshold** (*int*) – All nodes with degree lower than this value will be removed from the graph.

Returns **H** – A copy of the original graph or digraph with removed nodes.

Return type graph

evalne.utils.preprocess.**read_edge_embeddings**(*input_path*, *ebunch_len*, *embed_dim*, *delimiter=' '*, *method_name=None*)

Reads node-pair or edge embeddings from a file and returns the results as a matrix. Each line in the file is assumed to represent the embedding of one node-pair. File headers are inferred base on the expected number of embeddings and ignored.

Parameters

- **input_path** (*file or string*) – File or filename to read.
- **ebunch_len** (*int*) – The number of embeddings expected.
- **embed_dim** (*int*) – The expected dimensionality of the embeddings.
- **delimiter** (*string, optional*) – The string used to separate values in the input file. Default is ','.

- **method_name** (*string, optional*) – A string indicating the name of the method being evaluated. If provided will be used when logging an error. Default is None.

Returns **Y** – A column vector containing embeddings as rows.

Return type ndarray

Raises ValueError – If the number of embeddings in the file is less than ebunch_len. If the input data dimensions are not correct.

evalne.utils.preprocess.**read_labels**(*input_path*, *delimiter=','*, *comments='#'*, *idx_mapping=None*)

Reads node labels from a file and returns them as a numpy ndarray where the first column contains nodeIDs and the second, node attributes. If idx_mapping is provided, the original indices are re-mapped to new indices.

Parameters

- **input_path** (*file or string*) – File or filename to read. File is assumed to contain in each line a nodeID and label pair.
- **delimiter** (*string, optional*) – The string used to separate values in the input file. Default is ','.
- **comments** (*string, optional*) – The character used to indicate the start of a comment. Default is '#'.
- **idx_mapping** (*list of tuples*) – A list of (OldNodeID, NewNodeID).

Returns **labels** – A column vector of nodeID and label pairs.

Return type ndarray

evalne.utils.preprocess.**read_node_embeddings**(*input_path*, *nodes*, *embed_dim*, *delimiter=','*, *method_name=None*)

Reads node embeddings from a file (see Notes for expected input file structure) and returns the results as a dictionary. Keys correspond to nodeIDs and values to vectors. File headers are inferred base on the expected number of embeddings and ignored.

Parameters

- **input_path** (*file or string*) – File or filename to read.
- **nodes** (*array_like*) – The network nodes for which embeddings are expected.
- **embed_dim** (*int*) – The expected dimensionality of the node embeddings.
- **delimiter** (*string, optional*) – The string used to separate values in the input file. Default is ','.
- **method_name** (*string, optional*) – A string indicating the name of the method being evaluated. If provided will be used when logging an error. Default is None.

Returns **X** – A dictionary of {*nodeID*: embed_vect, *nodeID*: embed_vect, ... }. Keys are type string and values array_like.

Return type dict

Raises ValueError – If the number of embeddings in the file is less than len(nodes). If the input data dimensions are not correct.

Notes

The method accepts files where each line corresponds to the embedding of one node. The nodeID can be present as the first value in each line. If the nodeID is not present, embeddings are assumed to be written in the file in

ascending nodeID order.

evalne.utils.preprocess.**read_predictions**(*input_path*, *ebunch_len*, *delimiter=','*, *method_name=None*)

Reads node similarities from a file (see Notes for expected input file structure) and returns the results as a vector. File headers are inferred base on the expected number of predictions and ignored.

Parameters

- **input_path** (*file or string*) – File or filename to read.
- **ebunch_len** (*int*) – The number of predictions expected.
- **delimiter** (*string, optional*) – The string used to separate values in the input file. Default is ','.
- **method_name** (*string, optional*) – A string indicating the name of the method being evaluated. If provided will be used when logging an error. Default is None.

Returns **Y** – A column vector containing node similarities.

Return type ndarray

Raises ValueError – If the number of predictions in the file is less than ebunch_len. If the input data dimensions are not correct.

Notes

The method accepts files where all similarities are given in a single line in the file or one per line. In both case, the expected length of these "vectors" is ebunch_len. For input files containing similarities in rows, the method expects the last element of each row to be the similarity. This is useful for methods that return triplets: src, dst, sim.

evalne.utils.preprocess.**read_train_test**(*input_path*, *split*)

Reads the sets of train and test edges and non-edges from the given path that share the given split ID. The method assumes that these files follow the naming convention of store_train_test_splits().

Parameters

- **input_path** (*string*) – The path where the input data can be found.
- **split** (*int*) – The ID of the splits to be read.

Returns

- **train_E** (*set*) – Set of train edges
- **train_E_false** (*set*) – Set of train non-edges
- **test_E** (*set*) – Set of test edges
- **test_E_false** (*set*) – Set of test non-edges

See also:

evalne.utils.split_train_test.store_train_test_splits() The files in the provided input path are expected to follow the naming convention of this function.

evalne.utils.preprocess.**relabel_nodes**(*train_E*, *test_E*, *directed*)

For given sets of train and test edges, the method returns relabeled sets with nodes being integers in 0...N. Additionally, the method returns a graph containing all edges in the train and test and nodes in 0...N.

Parameters

- **train_E** (*set*) – The set of train edges.
- **test_E** (*set*) – The set of test edges.
- **directed** (*bool*) – Indicates if the method should return a graph or a digraph.

Returns

- **train_E** (*set*) – The set of train edges
- **test_E** (*set*) – The set of test edges
- **G** (*graph*) – A NetworkX graph or digraph with relabeled nodes containing the edges in train and test.
- **mapping** (*dict*) – A dictionary containing old node id's as key and new id's as values.

evalne.utils.preprocess.**save_graph**(*G*, *output_path*, *delimiter=', '*, *write_stats=True*, *write_weights=False*, *write_dir=True*)

Saves a graph to a file as an edgelist or weighted edgelist. If write_stats is True the file will include a header containing basic graph statistics as provided by the get_stats function.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **output_path** (*file or string*) – File or filename to write. If a file is provided, it must be opened in 'wb' mode.
- **delimiter** (*string, optional*) – The string to use for separating values in the output file. Default is ','.
- **write_stats** (*bool, optional*) – Adds basic graph statistics to the file as a header or not. Default is True.
- **write_weights** (*bool, optional*) – If True data will be stored as weighted edgelist i.e. triplets (src, dst, weight), otherwise, as regular (src, dst) pairs. For unweighted graphs, setting this parameter to True will add weight 1 to all edges. Default is False.
- **write_dir** (*bool, optional*) – This parameter is only relevant for undirected graphs. If True, it forces the method to write both edge directions in the file i.e. (src, dst) and (dst, src). If False, only one direction is stored. Default is True.

See also:

evalne.utils.split_train_test.get_stats() Function used to compute simple statistics to add as file header.

evalne.utils.split_train_test module

evalne.utils.split_train_test.**broder_alg**(*G*, *E*)

Runs Andrei Broder's algorithm[1] to select uniformly at random a spanning tree of the input graph. The method works for directed and undirected networks. The edge directions in the resulting spanning tree are taken from the input edgelist (E). For node pairs where both edge directions exist, one is chosen at random.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **E** (*set*) – A set of all directed or undirected edges in G.

[1] A. Broder, "Generating Random Spanning Trees", Proc. of the 30th Annual Symposium on Foundations of Computer Science, pp. 442–447, 1989.

Returns **train_E** – A set of edges of G that form the random spanning tree.

Return type set

See also:

evalne.utils.split_train_test.wilson_alg() A more computationally efficient approach for selecting a spanning tree.

References

evalne.utils.split_train_test.**check_overlap**(*filename*, *num_sets*)

Shows the amount of overlap (shared elements) between sets of node pairs with different split IDs. This function assumes the existance of num_sets files sharing the same name and with split IDs starting from 0 to num_sets.

Parameters

- **filename** (*string*) – Indicates the path and name (without split ID) of the first set. The sets are assumed to have sequential split IDs starting at 0.
- **num_sets** (*int*) – The number of sets for which to check the overlap.

See also:

evalne.utils.split_train_test.store_train_test_splits() The filename is expected to follow the naming convention of this function without "_<split_id>.csv".

Examples

Checking the overlap of several files in the test folder containing train edges:

```
>>> check_overlap("./test/tr_E", 2)
Intersection of 2 sets is 10
Union of 2 sets is 15
Jaccard coefficient: 0.666666666667
```

evalne.utils.split_train_test.**compute_splits_parallel**(*G*, *output_path*, *owa=True*, *train_frac=0.51*, *num_fe_train=None*, *num_fe_test=None*, *num_splits=10*)

Computes in parallel the required number of train/test splits of input graph edges. Also generated the same number of train and test non-edge sets. The resulting sets of node pairs are written to different files. The resulting train edge set has the following properties: spans a graph (digraph) with a single connected (weakly connected) component and the same nodes as G.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph with a single connected (weakly connected) component.
- **output_path** (*string*) – Indicates the path where data will be stored. Can include a name for all splits to share.
- **owa** (*bool, optional*) – Encodes the belief that the network respects or not the open world assumption. Default is True. If owa=True, train non-edges are sampled from the train graph only and can overlap with test edges. If owa=False, train non-edges are sampled from the full graph and cannot overlap with test edges.

- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.
- **num_fe_train** (*int, optional*) – The number of train non-edges to generate. Default is None (same number as train edges).
- **num_fe_test** (*int, optional*) – The number of test non-edges to generate. Default is None (same number as test edges).
- **num_splits** (*int, optional*) – The number of train/test splits to generate. Default is 10.

See also:

evalne.utils.split_train_test.split_train_test() This method used to split graph edges in train and test.

evalne.utils.split_train_test.generate_false_edges_owa() Method used to generate non-edges if owa=True.

evalne.utils.split_train_test.generate_false_edges_cwa() Method used to generate non-edges if owa=False.

evalne.utils.split_train_test.**generate_false_edges_cwa**(*G*, *train_E*, *test_E*, *num_fe_train=None*, *num_fe_test=None*)

Randomly samples pairs of unconnected nodes or non-edges from the input graph according to the closed world assumption (see Notes). Returns sets of train and test non-edges.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **train_E** (*set*) – The set of train edges.
- **test_E** (*set*) – The set of test edges.
- **num_fe_train** (*int, optional*) – The number of train non-edges to generate. Default is None (same number as train edges).
- **num_fe_test** (*int, optional*) – The number of test non-edges to generate. Default is None (same number as test edges).

Returns

- **train_E_false** (*set*) – The set of train non-edges.
- **test_E_false** (*set*) – The set of test non-edges.

Raises ValueError – If the input graph G has more than one (weakly) connected component. If more non-edges than existing in the graph are required.

Notes

The closed world assumption considers that one is certain about some node pairs being non-edges. Therefore, train non-edges cannot overlap with wither train not test edges. This scenario is common for static graphs where information about both the edges (positive interactions) and non-edges (negative interactions) is knows, e.g. protein protein interaction networks.

evalne.utils.split_train_test.**generate_false_edges_owa**(*G*, *train_E*, *test_E*, *num_fe_train=None*, *num_fe_test=None*)

Randomly samples pairs of unconnected nodes or non-edges from the input graph according to the open world assumption (see Notes). Returns sets of train and test non-edges.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **train_E** (*set*) – The set of train edges.
- **test_E** (*set*) – The set of test edges.
- **num_fe_train** (*int, optional*) – The number of train non-edges to generate. Default is None (same number as train edges).
- **num_fe_test** (*int, optional*) – The number of test non-edges to generate. Default is None (same number as test edges).

Returns

- **train_E_false** (*set*) – The set of train non-edges.
- **test_E_false** (*set*) – The set of test non-edges.

Raises ValueError – If the input graph G has more than one (weakly) connected component. If more non-edges than existing in the graph are required.

Notes

The open world assumption considers that for generating train non-edges one has access to the train edges only. Therefore, train non-edges could overlap with test edges. This scenario is common for evolving graphs where edges can only appear. In this case, one has access to certain train edges present in the graph at time t. Non-edges are then sampled using this information. At time t+1, the newly arrived test edges may have been previously selected as train non-edges. For undirected graphs the output is sorted (smallNodeID, bigNodeID).

evalne.utils.split_train_test.**naive_split_train_test**(*G*, *train_frac=0.51*)

Splits the edges of the input graph in sets of train and test and returns the results. Split is performed using the naive split approach (see Notes). The resulting train edge set has the following properties: spans a graph (digraph) with a single connected (weakly connected) component and the same nodes as G.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph with a single connected (weakly connected) component.
- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.

Returns

- **train_E** (*set*) – The set of train edges.
- **test_E** (*set*) – The set of test edges.

Raises ValueError – If the train_frac parameter is not in range (0, 1]. If the input graph G has more than one (weakly) connected component.

Notes

The method proceeds as follows: (1) compute the number of test edges needed from train_frac and size of G. (2) randomly remove edges from the input graph one at a time. If removing an edge causes the graph to become disconnected, add it back, otherwise, put it in the set of test edges. (3) repeat the previous step until all test edges are selected. (4) the remaining edges constitute the train set.

evalne.utils.split_train_test.**quick_nonedges**(*G*, *train_frac=0.51*, *fe_ratio=1.0*)

Randomly samples pairs of unconnected nodes or non-edges from the input graph according to the open world assumption. Is a more efficient implementation of *generate_false_edges_owa* which also does not require the train and test edge sets as input. Returns sets of train and test non-edges.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.
- **fe_ratio** (*float, optional*) – The ratio of non-edges to edges to sample. For fr_ratio > 0 and < 1 less non-edges than edges will be generated. For fe_edges > 1 more non-edges than edges will be generated. Default 1, same amounts.

Returns

- **train_E_false** (*ndarray*) – Column vector of train non-edges as pairs src, dst.
- **test_E_false** (*ndarray*) – Column vector of test non-edges as pairs src, dst.

Raises ValueError – If more non-edges than existing in the graph are required.

evalne.utils.split_train_test.**quick_split**(*G*, *train_frac=0.51*)

Splits the edges of the input graph in sets of train and test and returns the results. Split is performed using the quick split approach (see Notes). The resulting train edge set has the following properties: spans a graph (digraph) with a single connected (weakly connected) component and the same nodes as G.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph with a single connected (weakly connected) component.
- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.

Returns

- **train_E** (*ndarray*) – Column vector of train edges as pairs src, dst.
- **test_E** (*ndarray*) – Column vector of test edges as pairs src, dst.

Raises ValueError – If the train_frac parameter is not in range (0, 1]. If the input graph G has more than one (weakly) connected component.

Notes

The method proceeds as follows: (1) a spanning tree of the input graph is generated using a depth first tree approach starting at a random node, (2) randomly selected edges are added to those of the spanning tree until train_frac is reached, (3) the remaining edges, not used in previous steps, form the test set.

evalne.utils.split_train_test.**rand_split_train_test**(*G*, *train_frac=0.51*)

Splits the edges of the input graph in sets of train and test and returns the results. Split is performed using the

random split approach (see Notes). The resulting train edge set has the following properties: spans a graph (digraph) with a single connected (weakly connected) component.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.

Returns

- **train_E** (*set*) – The set of train edges.
- **test_E** (*set*) – The set of test edges.

Raises ValueError – If the train_frac parameter is not in range (0, 1].

Notes

The method proceeds as follows: (1) randomly remove 1-train_frac percent of edges from the input graph. (2) from the remaining edges compute the main connected component and these will be the train edges. (3) from the set of removed edges, those such that both end nodes exist in the train edge set computed in the previous step, are added to the final test set.

evalne.utils.split_train_test.**random_edge_sample**(*a*, *samp_frac=0.01*, *directed=False*, *sample_diag=False*)

Samples uniformly at random positions from the input adjacency matrix. The samples are returned in two sets, one containing all edges sampled and one containing all non-edges.

Parameters

- **a** (*sparse matrix*) – A sparse adjacency matrix.
- **samp_frac** (*float, optional*) – A float in range [0,1] representing the fraction of all edge pairs to sample. Default is 0.01 (1%)
- **directed** (*bool, optional*) – A flag indicating if the adjacency matrix should be considered directed or undirected. If undirected, indices are obtained only from the lower triangle. Default is False.
- **sample_diag** (*bool, optional*) – If True elements from the main diagonal are also sampled (i.e. self loops). Else they are excluded from the sampling process. Default is False.

Returns

- **pos_e** (*ndarray*) – Sampled node pairs that are edges.
- **neg_e** (*ndarray*) – Sampled node pairs that are non-edges.

evalne.utils.split_train_test.**redges_false**(*train_E*, *test_E*, *output_path=None*)

For directed graphs computes all non-edges (a->b) such that the opposite edge (a<-b) exists in the graph. It does this for both the train and test edge sets.

Parameters

- **train_E** (*set*) – The set of train edges.
- **test_E** (*set*) – The set of test edges.
- **output_path** (*string, optional*) – A path or file where to store the results. If None, results are not stored only returned. Default is None.

Returns

- **train_redges_false** (*set*) – A set of train edges respecting the mentioned property.
- **test_redges_false** (*set*) – A set of test edges respecting the mentioned property.

Notes

These non-edges can be used to asses the performance of the embedding methods on predicting non-reciprocated edges.

`evalne.utils.split_train_test.`**`split_train_test`**(*G*, *train_frac=0.51*, *st_alg='wilson'*)

Splits the edges of the input graph in sets of train and test and returns the results. Split is performed using the spanning tree approach (see Notes). The resulting train edge set has the following properties: spans a graph (digraph) with a single connected (weakly connected) component and the same nodes as G.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph with a single connected (weakly connected) component.
- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.
- **st_alg** (*string, optional*) – The algorithm to use for generating the spanning tree constituting the backbone of the train set. Options are: 'wilson' and 'broder'. The first option, 'wilson' is faster in most cases. Default is 'wilson'.

Returns

- **train_E** (*set*) – The set of train edges.
- **test_E** (*set*) – The set of test edges.

Raises `ValueError` – If the train_frac parameter is not in range (0, 1]. If the input graph G has more than one (weakly) connected component.

See also:

evalne.utils.split_train_test.broder_alg(), *evalne.utils.split_train_test.wilson_alg()*

Notes

The method proceeds as follows: (1) a spanning tree of the input graph is selected uniformly at random using broder or wilson's algorithm, (2) randomly selected edges are added to those of the spanning tree until train_frac is reached, (3) the remaining edges, not used in previous steps, form the test set.

`evalne.utils.split_train_test.`**`store_edgelists`**(*train_path*, *test_path*, *train_edges*, *test_edges*)

Writes the train and test edgelists to files with the specified names.

Parameters

- **train_path** (*string*) – Indicates the path where the train data will be stored.
- **test_path** (*string*) – Indicates the path where the test data will be stored.
- **train_edges** (*array_like*) – Set of train edges.
- **test_edges** (*array_like*) – Set of test edges.

evalne.utils.split_train_test.**store_train_test_splits**(*output_path*, *train_E*, *train_E_false*, *test_E*, *test_E_false*, *split_id=0*)

Writes the sets of train and test edges and non-edges to separate files in the provided path. All files will share the same split number as an identifier. If any folder in the path does not exist, it will be generated.

Parameters

- **output_path** (*string*) – Indicates the path where data will be stored.
- **train_E** (*set*) – Set of train edges.
- **train_E_false** (*set*) – Set of train non-edges.
- **test_E** (*set*) – Set of test edges.
- **test_E_false** (*set*) – Set of test non-edges.
- **split_id** (*int, optional*) – The ID of train/test split to be stored. Default is 0.

Returns **filenames** – A list of strings, the names and paths of the 4 files where the train and test edges and non-edges are stored.

Return type list

See also:

evalne.utils.preprocess.read_train_test() A function that can read the generated files.

Notes

This function generates 4 files under <output_path> named: 'trE_<split_id>.csv', 'negTrE_<split_id>.csv', 'teE_<split_id>.csv', 'negTeE_<split_id>.csv' corresponding respectively to train edges train non-edges, test edges and test non-edges.

Examples

Writes some data to files under a folder named test:

```
>>> train_E = ((1,2), (2,3))
>>> train_E_false = ((-1,-2), (-2,-3))
>>> test_E = ((10,20), (20,30))
>>> test_E_false = ((-10,-20), (-20,-30))
>>> store_train_test_splits("./test", train_E, train_E_false, test_E, test_E_
↪false, split_id=0)
('./test/trE_0.csv', './test/negTrE_0.csv', './test/teE_0.csv', './test/negTeE_0.
↪csv')
```

evalne.utils.split_train_test.**timestamp_split**(*G*, *train_frac=0.51*)

Splits the edges of the input graph in sets of train and test and returns the results. Split is performed using edge timestamps (see Notes). The resulting train edge set has the following properties: spans a graph (digraph) with a single connected (weakly connected) component.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph where edge weights are timestamps.
- **train_frac** (*float, optional*) – The proportion of train edges w.r.t. the total number of edges in the input graph (range (0.0, 1.0]). Default is 0.51.

Returns

- **train_E** (*ndarray*) – Column vector of train edges as pairs src, dst.
- **test_E** (*ndarray*) – Column vector of test edges as pairs src, dst.
- **tg** (*graph*) – A NetworkX graph containing only the edges in the train edge set.

Raises `ValueError` – If the train_frac parameter is not in range (0, 1]. If the input graph G has more than one (weakly) connected component.

Notes

The method proceeds as follows: (1) sort all edges by timestamp. (2) randomly remove 1-train_frac percent of edges from the input graph. (3) from the remaining edges compute the main connected component and these will be the train edges. (4) from the set of removed edges, those such that both end nodes exist in the train edge set computed in the previous step, are added to the final test set.

`evalne.utils.split_train_test.`**`wilson_alg`**(*G*, *E*)

Runs Willson's algorithm[2] also known as loop erasing random walk to select uniformly at random a spanning tree of the input graph. The method works for directed and undirected networks[3]. The edge directions in the resulting spanning tree are taken from the input edgelist (E). For node pairs where both edge directions exist, one is chosen at random.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **E** (*set*) – A set of all directed or undirected edges in G.

Returns **train_E** – A set of edges of G that form the random spanning tree.

Return type set

See also:

evalne.utils.split_train_test.broder_alg() A different approach for selecting a spanning tree that could be faster for specific graphs.

References

evalne.utils.util module

exception `evalne.utils.util.`**`TimeoutExpired`**

Bases: `Exception`

`evalne.utils.util.`**`auto_import`**(*classpath*)

Imports any Sklearn binary classifier from a string.

Parameters **classpath** (*string*) – A string indicating the full path the any Sklearn classifier and its parameters.

Returns **clf** – The classifier instance.

Return type object

Raises `ValueError` – If the classifier could not be imported.

[2] D. B. Wilson, "Generating Random Spanning Trees More Quickly than the Cover Time", In Proceedings of STOC, pp. 296–303, 1996.

[3] J. G. Propp and D. B. Wilson, "How to Get a Perfectly Random Sample from a Generic Markov Chain and Generate a Random Spanning Tree of a Directed Graph", Journal of Algorithms 27, pp. 170–217, 1998.

Examples

Importing the SVC classifier with user-defined parameters:

```
>>> auto_import("sklearn.svm.SVC(C=1.0, kernel='rbf')")
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Importing a decision tree classifier with no parameters:

```
>>> auto_import("sklearn.ensemble.ExtraTreesClassifier()")
ExtraTreesClassifier(bootstrap=False, class_weight=None, criterion='gini',
    max_depth=None, max_features='auto', max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
    oob_score=False, random_state=None, verbose=0, warm_start=False)
```

evalne.utils.util.**run**(*cmd*, *timeout*, *verbose*)

Runs the cmd command provided as input in a new process. If execution time exceeds timeout, the process is killed and a TimeoutExpired exception is raised.

Parameters

- **cmd** (*string*) – A string indicating the command to run on the command line.
- **timeout** (*int or float*) – A value indicating the maximum number of second the process is allowed to run for.
- **verbose** (*bool*) – Boolean indicating if the execution output should be shown or not (pipes stdout and stderr to devnull).

Raises *TimeoutExpired* – If the execution time exceeds the number of second indicated by timeout.

Notes

The method additionally raises ImportError, IOError and AttributeError if these are encountered during execution of the cmd command.

Examples

Runs a command that prints Start, sleeps for 5 seconds and prints Done

```
>>> util.run("python -c 'import time; print("Start"); time.sleep(5); print("Done")
↪'", 7, True)
Start
Done
```

Same as previous command but now it does not print Done because it times out after 2 seconds

```
>>> util.run("python -c 'import time; print("Start"); time.sleep(5); print("Done")
↪'", 2, True)
Start
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<input>", line 1, in <module>
  File "EvalNE/evalne/utils/util.py", line 84, in run
    A string indicating the command to run on the command line.
TimeoutExpired: Command `python -c 'import time; print("Start"); time.sleep(5);␣
↪print("Done")'` timed out
```

evalne.utils.util.**run_function**(*timeout*, *func*, **args*)

Runs the function provided as input in a new process. If execution time exceeds timeout, the process is killed and a TimeoutExpired exception is raised.

Parameters

- **timeout** (*int or float*) – A value indicating the maximum number of seconds the process is allowed to run for or None.
- **func** (*object*) – A function to be executed. First parameter must be a queue object. Check notes section for more details.
- ***args** – Variable length argument list for function func. The list shloud **not** include the queue object.

Raises *TimeoutExpired* – If the execution time exceeds the number of second indicated by timeout.

Notes

The input function func must take as a first parameter a queue object q. This is used to communicate results between the process where the function is running and the main thread. The list of args does not need to include the queue object, it is automatically inserted by this function.

Examples

Runs function foo for at most 100 seconds and returns the result. Foo must put the result in q.

```
>>> def foo(q, a, b):
...     q.put(a+b)
...
>>> run_function(100, foo, *[1, 2])
3
```

evalne.utils.viz_utils module

evalne.utils.viz_utils.**parallel_coord**(*scoresheet*, *features*, *class_col='methods'*)

Generates a parallel coordinate plot from the given Scoresheet object and the set of features specified.

Parameters

- **scoresheet** (*evalne.Scoresheet*) – A Scoresheet object containing the results of an evaluation.
- **features** (*list*) – A list of strings indicating the features to show in the plot (in addition to methods and networks). Accepted features are: 'auroc', 'average_precision', 'precision', 'recall', 'fallout', 'miss', 'accuracy', 'f_score', *eval_time* and *edge_embed_method*.

- **class_col** (*string, optional*) – Indicates the class to highlight. Options are *methods* and *networks*. Default is *methods*.

evalne.utils.viz_utils.**plot_curve**(*filename*, *x*, *y*, *x_label*, *y_label*, *title=None*)

Plots y coordinates against x coordinates as a line.

Parameters

- **filename** (*string*) – A file or filename where to store the plot.
- **x** (*array_like*) – The x coordinates of the plot.
- **y** (*array_like*) – The y coordinates of the plot.
- **x_label** (*string*) – The label of the x axis.
- **y_label** (*string*) – The label of the y axis.
- **title** (*string or None, optional*) – The title of the plot. Default is None (no title).

evalne.utils.viz_utils.**plot_emb2d**(*emb*, *colors=None*, *filename=None*)

Generates a scatter plot of the given embeddings. Optional colors for the nodes can be provided as well as a filename to store the results. If dim of embeddings > 2, uses t-SNE to reduce it to 2.

Parameters

- **emb** (*matrix*) – A Numpy matrix containing the node or edge embeddings.
- **colors** (*array, optional*) – A Numpy array containing the colors of each node. Default is None.
- **filename** (*string, optional*) – A string indicating the path and name of the file where to store the scatter plot. If not provided the plot is shown on screen. Default is None.

evalne.utils.viz_utils.**plot_graph2d**(*G*, *emb=None*, *labels=None*, *colors=None*, *filename=None*)

Plots the given graph in 2d. If the embeddings of nodes are provided, they are used to place the nodes on the 2d plane. If dim of embeddings > 2, then its reduced to 2 using t-SNE. Optional labels and colors for the nodes can be provided, as well as a filename to store the results.

Parameters

- **G** (*graph*) – A NetworkX graph or digraph.
- **emb** (*matrix, optional*) – A Numpy matrix containing the node embeddings. Default is None.
- **labels** (*dict, optional*) – A dictionary containing nodeIDs as keys and node labels as values. Default is None.
- **colors** (*array, optional*) – A Numpy array containing the colors of each graph node. Default is None.
- **filename** (*string, optional*) – A string indicating the path and name of the file where to store the scatter plot. If not provided the plot is showed on screen. Default is None.

Module contents

4.1.2 Module contents

EvalNE

EvalNE is a Python package for the evaluation of network embedding methods on a variety of downstream prediction tasks. These tasks include link prediction, sign prediction, network reconstruction and node classification. Basic embedding and graph visualization functions are also provided.

See https://evalne.readthedocs.io/en/latest/ for complete documentation.

CHAPTER 5

Release Log

5.1 EvalNE v0.4.0

Release date: 04 Jul 2022

5.1.1 Documentation

- Release log update.
- Docstring improvements affecting some classes and functions.
- Improved Readme.md and docs files.
- Improved conf.ini inline documentation.

5.1.2 Miscellaneous

- Transition to python >3.7.
- Support for python 2 has been dropped.
- Dependencies bumped to latest package versions including numpy and networkx.
- Improved imports throughout the library.
- Extended parameter checks for config files.

5.2 EvalNE v0.3.4

Release date: 05 Dec 2022

5.2.1 Documentation

- Release log update.
- Docstring improvements affecting some classes and functions.
- Improved Readme.md and docs files.

5.2.2 New features

- Included a bash script to control the number of threads used by numpy during execution.
- Included a new label binarization method ('prop') which binarizes predictions based on the number of positive/negative instances in the train data.
- The library now logs the logistic regression coefficients per method when LR or LRCV are used.
- Included a new performance metric, average precision for the LP, SP, and NR tasks.
- Parameter checks in the EvalSetup class for .ini configuration files can now be turned on or off.
- New parallel coordinates plot has been added to visualize method performance from output pickle files.

5.2.3 Miscellaneous

- Input type errors are now cached and logged and no longer cause the evaluation to crash.

5.3 EvalNE v0.3.3

Release date: 14 Dec 2020

5.3.1 Documentation

- Release log update.
- Extensive docstring improvements affecting all classes and functions.
- Docstring examples included for all important classes and low level functions.
- Improved variable descriptions in conf.ini.
- Improved Readme.md and docs files.

5.3.2 New features

- Sign prediction added as a downstream task that can be evaluated (using the SPEvaluator class).
- Three new classes (LPEvalSplit, SPEvalSplit and NREvalSplit) added that simplify the computation of evaluation splits for the LP, SP and NR downstream tasks.
- Added three new heuristic baselines: Cosine Similarity, Leicht-Holme-Newman index and Topological Overlap.
- When used as a command line tool, the library now provides both the train and test evaluation scores in the output folder.
- When used as an API the user can now conveniently store the model predictions for any downstream task.

- Added timeout for baselines evaluation
- Added function that can run other functions in a separate process with given timeout.

5.3.3 Miscellaneous

- Improved requirement specification in requirements.txt and setup.py.
- Improved library and module level imports.
- General improvements on error and warning messages.
- Memory errors are now catched and logged.
- All numerical output is now rounded to 4 decimals.

5.3.4 Bugs

- Fixed a bug that would cause a TimeoutError to be raised incorrectly.

5.4 EvalNE v0.3.2

Release date: 10 Dec 2019

5.4.1 Documentation

- Release log update
- Various docstring improvements
- Improved variable descriptions in conf.ini

5.4.2 New features

- The user can now set a timeout for the execution of each method in the conf files. E.g. TIMEOUT = 1800
- Conf files now support any sklearn binary classifer in the LP_MODEL variable. E.g. LP_MODEL=sklearn.svm.LinearSVC(C=1.0, kernel='rbf', degree=3)
- Conf files also support keyword SVM for the LP_MODEL. This uses the sklearn LinearSVC model and tunes the regularization parameter on a grid [0.1, 1, 10, 100, 1000].
- Method execution is made safer by using Popen communicate instead of subprocess.run(shell=True)
- Removed lp_model coefficient output. This could lead to errors and failed evaluations for certain Sklearn binary classifiers
- Method compute_pred() of LPEvaluator and NREvaluator classes now tries to use lp_model.predict_proba() if the classifier does not have it, the function defaults to lp_model.predict()
- The scoresheet method get_pandas_df() now includes a repeat parameter which denotes the exact experiment repeat results the user wants in the DF. If repeat=None, the DF returned will contain the average metric over all experiment repeats.

5.4.3 Miscellaneous

- Log file output now shows timeout errors and LR method selected
- Corrected the cases where some warnings were reported as errors
- Added util.py in the utils module

5.4.4 Bugs

- Fixed bug which would prevent the library to store the output when executed from Py3

5.5 EvalNE v0.3.1

Release date: 2 Nov 2019

5.5.1 Documentation

- Release log update
- Various docstring improvements

5.5.2 New features

- New heuristic for LP named *all_baselines*. Generates a 5-dim edge embedding by combining the existing heuristics [CN, JC, AA, PA, RAI].
- Automated file headder detection (in the output of embedding methods) is now a function
- Functions for reading the embeddings, predictions and node labels have been added

5.5.3 Miscellaneous

- General improvements in NC task
- Added NCScores and NCResults classes
- Pickle file containig evaluation results is now saved incrementally, after each networks has been evaluated. If the user stops the process mid-way the results up to the last network will be available
- Coefficients of the binary classifier per evaluated method are now provided for LP and NR tasks
- Improved exception management
- Improved conf file sanity checks
- Evaluated methods now return a single Results object instead of a list

5.5.4 Bugs

- Fixed bug related to plotting PR and AUC curves
- Fixed node classification bugs preventing the evaluaition to run properly

5.6 EvalNE v0.3.0

Release date: 21 Oct 2019

5.6.1 Documentation

- Release log update

5.6.2 New features

- Old Evaluator class is now LPEvaluator
- Added Network Reconstruction evaluation (NREvaluator)
- Added Node Classification evaluation (NCEvaluator)
- Train/validation splits are now required when initializing Evaluator classes
- Added 3 new algorithms for computing train/test splits. One extremely scalable up to millions of nodes/edges
- Improved error management and error logging
- Edge embedding methods are now always tunned as method parameters. Results for the best are given.
- For link prediction and network reconstruction the user can now evaluate the methods exclusively on train data.
- Addes Scoresheet class to simplify output management
- Export results directly to pandas dataframe and latex tables suppored

5.6.3 Miscellaneous

- Changed default parameters for EvalSplit
- Added new parameter for EvalSplit.set_split()
- Evaluation output is now always stored as pickle file
- Execution time per method and dataset is not provided
- Train/test average time per dataset is registered
- Added *auto* mode for the Results class to decide if train or test data should be logged

5.7 EvalNE v0.2.3

Release date: 25 Apr 2019

5.7.1 Documentation

- Release log update
- Library diagram minor update

5.7.2 Bugs

- Corrected parameter tuning rutine which was minimizing the objective metric given instead of maximizing it.
- Corrected evaluate_cmd() function output.

5.7.3 New features

- Evaluation output file now contains also a table of execution times per evaluated method.

5.7.4 Miscellaneous

- Changed behaviour of verbosity flag. Now, if Verbose=False it deactivates all stdout for the methods being evaluated (not stderr) but maintains the library stdout.
- Added more conf.ini files for reproducing the experimental section of different papers.

5.8 EvalNE v0.2.2

Release date: 14 Mar 2019

5.8.1 Documentation

- Readme and docs update to include pip installation

5.8.2 Miscelaneous

- Library is now pip installable
- Minor bugfixes

5.9 EvalNE v0.2.1

Release date: 13 Mar 2019

5.9.1 New features

- Added *WRITE_WEIGHTS_OTHER* in conf files which allows the user to specify if the input train network to the NE methods should have weights or not. If True but the original input network is unweighted, weights of 1 are given to each edge. This features is useful for e.g. the original code of LINE, which requires edges to have weights (all 1 if the graph is unweighted).
- Added *WRITE_DIR_OTHER* in conf files which allows the user to specify if the input train network to the NE methods should be specified with both directions of edges or a single one.
- Added *SEED* in the conf file which sets a general random seed for the whole library. If None the system time is used.
- Added a faster method for splitting non-edges in train and test when all non-edges in the graph are required.

5.9.2 Documentation

- Readme and docs update
- Descriptions of each option in conf.ini added

5.9.3 Miscellaneous

- Removed optional seed parameter from all methods in split_train_test.py
- Removed random seed resetting in the edges split methods
- *simple-example.py* now checks if OpenNE is installed, if not it runs only the LP heuristics.
- Sklearn removed from requirements.txt (already satisfied by scikit-learn)
- *setup.py* update. Ready for making EvalNE pip installable.
- Train/validation fraction was 50/50 which caused the train set to be excesively small and parameter validation not accurate. New value is 90/10.
- Improved warnings in evaluator code
- General code cleaning

5.9.4 Bugs

- train/validation and train/test splits used the same random seed for generating the edge split which caused correlation between them. Now the train/validation split is random.
- Fixed a bug which would cause the evaluation of any edge embedding method to crash.
- Precitions from edge embeddings were computed using LogisticRegression.predict(). This gives class labels and not class probabilities resulting in worst estimates of method performance. This has been changed to LogisticRegression.predict_proba()

5.10 EvalNE v0.2.0

Release date: 4 Feb 2019

5.10.1 API changes

- The evaluate_ne_cmd method has been renamed to evaluate_cmd
- evaluate_cmd can now evaluate node, edge or end to end embedding method
- evaluate_cmd a new method_type parameter has been added to indicate how the method should be evaluated (ne, ee or e2e)
- ScoreSheet object has been removed
- Score method removed from Katz and KatzApprox classes
- Method get_parameters() from Evaluator has been removed

5.10.2 New features

- Added method_type option in *.ini* files to evaluate (ne, ee or e2e)
- compute_results method now takes an optional label binarizer parameter
- evaluate_ne method now takes an optional label binarizer parameter
- save and pretty_print methods in Results now take a precatk_vals parameter which indcates for which k values to compute this score
- When REPORT SCORES = all is selected in the *.ini* file, the library now presents all the available metrics for each algorithm and dataset averaged over the number of repetitions.

5.10.3 Documentation

- Docstring updates
- Release log added to Docs
- Contributing added to Docs

5.10.4 Miscellaneous

- Exception handling improvements

5.10.5 Bugs

- Prevented possible infinite loop while generating non-edges by raising a warning if the used-selected values is > that the max possible non-edges.

CHAPTER 6

Contributing

Contributions in order to improve or extend the capabilities of EvalNE are highly appreciated. There are different ways in which interested users can contribute to the library, these include:

- Raising issues on GitHub
  - Reporting bugs
  - Requesting new features
  - Questions
- Pull requests for new features on GitHub
- Improving the documentation

CHAPTER 7

License

CHAPTER 8

Acknowledgements

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant Agreement no. 615517, from the FWO (project no. G091017N, G0F9816N), and from the European Union's Horizon 2020 research and innovation programme and the FWO under the Marie Sklodowska-Curie Grant Agreement no. 665501.

Research Foundation
Flanders
Opening new horizons

CHAPTER 9

Help

For help with setting up or using EvalNE plase contact: alexandru(dot)mara(at)ugent(dot)be

CHAPTER 10

Citation

If you have found EvaNE useful in your research, please cite our arXiv paper :

```
@article{MARA2022evalne,
  title = {EvalNE: A Framework for Network Embedding Evaluation},
  author = {Alexandru Mara and Jefrey Lijffijt and Tijl {De Bie}},
  journal = {SoftwareX},
  volume = {17},
  pages = {},
  year = {2022},
  issn = {100997},
  doi = {10.1016/j.softx.2022.100997},
  url = {https://www.sciencedirect.com/science/article/pii/S2352711022000139}
}
```

Python Module Index

e

evalne, 76
evalne.evaluation, 54
evalne.evaluation.edge_embeddings, 13
evalne.evaluation.evaluator, 17
evalne.evaluation.pipeline, 33
evalne.evaluation.score, 37
evalne.evaluation.split, 47
evalne.methods, 60
evalne.methods.katz, 54
evalne.methods.similarity, 55
evalne.utils, 76
evalne.utils.preprocess, 61
evalne.utils.split_train_test, 65
evalne.utils.util, 73
evalne.utils.viz_utils, 75

Index

A

accuracy() (*evalne.evaluation.score.Scores method*), 42
adamic_adar_index() (*in module evalne.methods.similarity*), 58
all_baselines() (*in module evalne.methods.similarity*), 60
auroc() (*evalne.evaluation.score.Scores method*), 42
auto_import() (*in module evalne.utils.util*), 73
average() (*in module evalne.evaluation.edge_embeddings*), 13
average_precision() (*evalne.evaluation.score.Scores method*), 43

B

BaseEvalSplit (*class in evalne.evaluation.split*), 47
broder_alg() (*in module evalne.utils.split_train_test*), 65

C

check_overlap() (*in module evalne.utils.split_train_test*), 66
comments (*evalne.evaluation.pipeline.EvalSetup attribute*), 34
common_neighbours() (*in module evalne.methods.similarity*), 55
compute_edge_embeddings() (*in module evalne.evaluation.edge_embeddings*), 14
compute_ee() (*evalne.evaluation.evaluator.LPEvaluator method*), 18
compute_pred() (*evalne.evaluation.evaluator.LPEvaluator method*), 18
compute_pred() (*evalne.evaluation.evaluator.NCEvaluator method*), 24
compute_results() (*evalne.evaluation.evaluator.LPEvaluator method*), 19
compute_results() (*evalne.evaluation.evaluator.NCEvaluator method*), 25
compute_splits() (*evalne.evaluation.split.LPEvalSplit method*), 49
compute_splits() (*evalne.evaluation.split.NREvalSplit method*), 51
compute_splits() (*evalne.evaluation.split.SPEvalSplit method*), 52
compute_splits_parallel() (*in module evalne.utils.split_train_test*), 66
cosine_similarity() (*in module evalne.methods.similarity*), 56
curves (*evalne.evaluation.pipeline.EvalSetup attribute*), 34

D

del_selfloops (*evalne.evaluation.pipeline.EvalSetup attribute*), 34
delimiter (*evalne.evaluation.pipeline.EvalSetup attribute*), 34
directed (*evalne.evaluation.pipeline.EvalSetup attribute*), 34

E

edge_embedding_methods (*evalne.evaluation.pipeline.EvalSetup attribute*), 34
embed_dim (*evalne.evaluation.pipeline.EvalSetup attribute*), 34
embtype_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 34
evalne (*module*), 76
evalne.evaluation (*module*), 54
evalne.evaluation.edge_embeddings (*module*), 13
evalne.evaluation.evaluator (*module*), 17
evalne.evaluation.pipeline (*module*), 33
evalne.evaluation.score (*module*), 37
evalne.evaluation.split (*module*), 47
evalne.methods (*module*), 60

evalne.methods.katz (*module*), 54
evalne.methods.similarity (*module*), 55
evalne.utils (*module*), 76
evalne.utils.preprocess (*module*), 61
evalne.utils.split_train_test (*module*), 65
evalne.utils.util (*module*), 73
evalne.utils.viz_utils (*module*), 75
EvalSetup (*class in evalne.evaluation.pipeline*), 33
EvalSplit (*class in evalne.evaluation.split*), 48
evaluate_baseline() (*evalne.evaluation.evaluator.LPEvaluator method*), 19
evaluate_baseline() (*evalne.evaluation.evaluator.SPEvaluator method*), 32
evaluate_cmd() (*evalne.evaluation.evaluator.LPEvaluator method*), 20
evaluate_cmd() (*evalne.evaluation.evaluator.NCEvaluator method*), 25
evaluate_cmd() (*evalne.evaluation.evaluator.NREvaluator method*), 28
evaluate_ne() (*evalne.evaluation.evaluator.LPEvaluator method*), 23
evaluate_ne() (*evalne.evaluation.evaluator.NCEvaluator method*), 27

F

f1_macro() (*evalne.evaluation.score.NCScores method*), 39
f1_micro() (*evalne.evaluation.score.NCScores method*), 39
f1_weighted() (*evalne.evaluation.score.NCScores method*), 39
f_score() (*evalne.evaluation.score.Scores method*), 43
fallout() (*evalne.evaluation.score.Scores method*), 43
fe_ratio (*evalne.evaluation.pipeline.EvalSetup attribute*), 34
fe_ratio (*evalne.evaluation.split.LPEvalSplit attribute*), 50
fit_predict() (*evalne.methods.katz.KatzApprox method*), 55

G

generate_false_edges_cwa() (*in module evalne.utils.split_train_test*), 67
generate_false_edges_owa() (*in module evalne.utils.split_train_test*), 67
get_all() (*evalne.evaluation.score.NCResults method*), 38
get_all() (*evalne.evaluation.score.Results method*), 41
get_data() (*evalne.evaluation.split.BaseEvalSplit method*), 47
get_latex() (*evalne.evaluation.score.Scoresheet method*), 44
get_pandas_df() (*evalne.evaluation.score.Scoresheet method*), 44
get_parameters() (*evalne.evaluation.split.BaseEvalSplit method*), 47
get_parameters() (*evalne.evaluation.split.LPEvalSplit method*), 50
get_parameters() (*evalne.evaluation.split.NREvalSplit method*), 52
get_params() (*evalne.methods.katz.Katz method*), 54
get_params() (*evalne.methods.katz.KatzApprox method*), 55
get_redges_false() (*in module evalne.utils.preprocess*), 61
get_stats() (*in module evalne.utils.preprocess*), 61
getboollist() (*evalne.evaluation.pipeline.EvalSetup method*), 34
getlinelist() (*evalne.evaluation.pipeline.EvalSetup method*), 34
getlist() (*evalne.evaluation.pipeline.EvalSetup method*), 35
getseplist() (*evalne.evaluation.pipeline.EvalSetup method*), 35
gettuneparams() (*evalne.evaluation.pipeline.EvalSetup method*), 35

H

hadamard() (*in module evalne.evaluation.edge_embeddings*), 14

I

infer_header() (*in module evalne.utils.preprocess*), 61
inpaths (*evalne.evaluation.pipeline.EvalSetup attribute*), 35
input_delim_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 35

J

jaccard_coefficient() (*in module evalne.methods.similarity*), 56

K

Katz (*class in evalne.methods.katz*), 54
KatzApprox (*class in evalne.methods.katz*), 54

L

labelpaths (*evalne.evaluation.pipeline.EvalSetup attribute*), 35

lhn_index() (*in module evalne.methods.similarity*), 57
load_graph() (*in module evalne.utils.preprocess*), 61
log_results() (*evalne.evaluation.score.Scoresheet method*), 45
lp_baselines (*evalne.evaluation.pipeline.EvalSetup attribute*), 35
lp_model (*evalne.evaluation.pipeline.EvalSetup attribute*), 35
lp_num_edge_splits (*evalne.evaluation.pipeline.EvalSetup attribute*), 35
LPEvalSplit (*class in evalne.evaluation.split*), 49
LPEvaluator (*class in evalne.evaluation.evaluator*), 17

M

maximize (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
methods_opne (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
methods_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
miss() (*evalne.evaluation.score.Scores method*), 43

N

naive_split_train_test() (*in module evalne.utils.split_train_test*), 68
names (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
names_opne (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
names_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
nc_node_fracs (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
nc_num_node_splits (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
NCEvaluator (*class in evalne.evaluation.evaluator*), 23
NCResults (*class in evalne.evaluation.score*), 37
NCScores (*class in evalne.evaluation.score*), 39
neighbourhood (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
nr_edge_samp_frac (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
NREvalSplit (*class in evalne.evaluation.split*), 51
NREvaluator (*class in evalne.evaluation.evaluator*), 27
nw_name (*evalne.evaluation.split.BaseEvalSplit attribute*), 47

O

output_delim_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
output_format_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
owa (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
owa (*evalne.evaluation.split.LPEvalSplit attribute*), 50

P

parallel_coord() (*in module evalne.utils.viz_utils*), 75
plot() (*evalne.evaluation.score.Results method*), 41
plot_curve() (*in module evalne.utils.viz_utils*), 76
plot_emb2d() (*in module evalne.utils.viz_utils*), 76
plot_graph2d() (*in module evalne.utils.viz_utils*), 76
precatk_vals (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
precision() (*evalne.evaluation.score.Scores method*), 43
precisionatk() (*evalne.evaluation.score.Scores method*), 43
predict() (*evalne.methods.katz.Katz method*), 54
preferential_attachment() (*in module evalne.methods.similarity*), 59
prep_graph() (*in module evalne.utils.preprocess*), 62
pretty_print() (*evalne.evaluation.score.NCResults method*), 38
pretty_print() (*evalne.evaluation.score.Results method*), 41
print_tabular() (*evalne.evaluation.score.Scoresheet method*), 45
prune_nodes() (*in module evalne.utils.preprocess*), 62

Q

quick_nonedges() (*in module evalne.utils.split_train_test*), 69
quick_split() (*in module evalne.utils.split_train_test*), 69

R

rand_split_train_test() (*in module evalne.utils.split_train_test*), 69
random_edge_sample() (*in module evalne.utils.split_train_test*), 70
random_prediction() (*in module evalne.methods.similarity*), 60
read_edge_embeddings() (*in module evalne.utils.preprocess*), 62
read_labels() (*in module evalne.utils.preprocess*), 63

read_node_embeddings() (*in module evalne.utils.preprocess*), 63
read_predictions() (*in module evalne.utils.preprocess*), 64
read_splits() (*evalne.evaluation.split.EvalSplit method*), 49
read_train_test() (*in module evalne.utils.preprocess*), 64
recall() (*evalne.evaluation.score.Scores method*), 44
redges_false() (*in module evalne.utils.split_train_test*), 70
relabel (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
relabel_nodes() (*in module evalne.utils.preprocess*), 64
resource_allocation_index() (*in module evalne.methods.similarity*), 59
Results (*class in evalne.evaluation.score*), 40
run() (*in module evalne.utils.util*), 74
run_function() (*in module evalne.utils.util*), 75

S

samp_frac (*evalne.evaluation.split.NREvalSplit attribute*), 52
save() (*evalne.evaluation.score.NCResults method*), 38
save() (*evalne.evaluation.score.Results method*), 41
save_graph() (*in module evalne.utils.preprocess*), 65
save_predictions() (*evalne.evaluation.score.NCResults method*), 39
save_predictions() (*evalne.evaluation.score.Results method*), 42
save_prep_nw (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
save_sim_matrix() (*evalne.methods.katz.Katz method*), 54
save_tr_graph() (*evalne.evaluation.split.BaseEvalSplit method*), 48
Scores (*class in evalne.evaluation.score*), 42
scores (*evalne.evaluation.pipeline.EvalSetup attribute*), 36
Scoresheet (*class in evalne.evaluation.score*), 44
seed (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
separators (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
set_splits() (*evalne.evaluation.split.LPEvalSplit method*), 50
set_splits() (*evalne.evaluation.split.NREvalSplit method*), 52
set_splits() (*evalne.evaluation.split.SPEvalSplit method*), 53
SPEvalSplit (*class in evalne.evaluation.split*), 52
SPEvaluator (*class in evalne.evaluation.evaluator*), 31
split_alg (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
split_alg (*evalne.evaluation.split.BaseEvalSplit attribute*), 48
split_id (*evalne.evaluation.split.BaseEvalSplit attribute*), 48
split_train_test() (*in module evalne.utils.split_train_test*), 71
store_edgelists() (*evalne.evaluation.split.BaseEvalSplit method*), 48
store_edgelists() (*in module evalne.utils.split_train_test*), 71
store_train_test_splits() (*in module evalne.utils.split_train_test*), 71

T

task (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
test_edges (*evalne.evaluation.split.BaseEvalSplit attribute*), 48
test_labels (*evalne.evaluation.split.BaseEvalSplit attribute*), 48
TG (*evalne.evaluation.split.BaseEvalSplit attribute*), 47
timeout (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
TimeoutExpired, 73
timestamp_split() (*in module evalne.utils.split_train_test*), 72
topological_overlap() (*in module evalne.methods.similarity*), 57
train_edges (*evalne.evaluation.split.BaseEvalSplit attribute*), 48
train_frac (*evalne.evaluation.split.BaseEvalSplit attribute*), 48
train_labels (*evalne.evaluation.split.BaseEvalSplit attribute*), 48
traintest_frac (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
trainvalid_frac (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
tune_params_opne (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
tune_params_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 37

V

verbose (*evalne.evaluation.pipeline.EvalSetup attribute*), 37

W

weighted_l1() (*in module evalne.evaluation.edge_embeddings*), 15
weighted_l2() (*in module evalne.evaluation.edge_embeddings*), 16
wilson_alg() (*in module evalne.utils.split_train_test*), 73
write_all() (*evalne.evaluation.score.Scoresheet method*), 46
write_dir_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
write_pickle() (*evalne.evaluation.score.Scoresheet method*), 47
write_stats (*evalne.evaluation.pipeline.EvalSetup attribute*), 37
write_tabular() (*evalne.evaluation.score.Scoresheet method*), 47
write_weights_other (*evalne.evaluation.pipeline.EvalSetup attribute*), 37